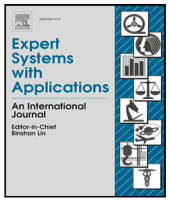
Expert Systems With Applications 254 (2024) 124346

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa



Orchestrating scheduling, grouping and parallelism to enhance the performance of distributed stream computing system

Dawei Sun [a,*], Haiyang Chen [a], Shang Gao [b], Rajkumar Buyya [c]

[a] School of Information Engineering, China University of Geosciences, Beijing, 100083, China
[b] School of Information Technology, Deakin University, Waurn Ponds, Victoria 3216, Australia
[c] Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia

ARTICLE INFO

Keywords:
Stream application scheduling
Stream grouping
Operator parallelism
Stream computing
Big data

ABSTRACT

In a big data stream computing environment, the arrival rate of data streams usually fluctuates over time, posing a great challenge to the elasticity of system. The performance of stream computing system is crucial, especially when dealing with unbounded and fluctuating data streams. Most prior studies have primarily focused on one or two aspects to enable elasticity, often lacking prompt and comprehensive performance optimization. This limitation could lead to a tuning bottleneck, preventing the system's performance from consistently reaching its optimal state. Additionally, many stream computing systems are not intelligently adaptive in real time due to the challenges of manual parameter reconfiguration for fluctuating streams. To better address these issues, we propose a framework named Sgp-Stream, which orchestrates scheduling, grouping and parallelism (Sgp). To enhance the system performance. We conduct the following research: (1) Running experiments to evaluate the impact of different factors such as scheduling, grouping and parallelism on system performance. Results show that factors at a single level usually have an upper limit on tuning system performance, and better overall performance can be achieved by coordinating multi-level factors. (2) Establishing quantitative models for stream application that consider computational cost and communication cost, multi-dimensional featured data stream, data center resources, and latency & throughput performance. (3) Demonstrating the effectiveness of the proposed runtime-aware data stream grouping based on smooth weighted polling, elastic adaptive scheduling based on Linear Deterministic Greedy and elastic scaling strategy based on Gradient Descent in Sgp-Stream, for continuous performance optimization.(4) Evaluating the application latency, throughput and resource utilization objectives using a real-world elastic stream computing system and twitter data set. Experimental results show that, compared to existing state-of-the-art works, the proposed Sgp-Stream outperforms them by reducing latency by 26%–48%, improving throughput by 14%–20%, and increasing resource utilization rate by 15%–21%, especially under increasing data stream input rates.

1. Introduction

Large volume of continuous data have brought great challenges to stream computing systems, particularly in terms of system performance during operation. Stream computing systems can be applied in various emerging scenarios (Ramesh et al., 2021), such as monitoring infrastructure sensors (Sun et al., 2019), intelligent household (Cao et al., 2020) and smart healthcare (García-Vico et al., 2021). In these applications, data can be produced at incredibly high rates, reaching up to 40 billion bits per second (bps), demanding real-time processing (Herodotou et al., 2022). Nowadays, research is extensive, and some streaming computing frameworks efficiently process and analyze real-time data streams by utilizing the combined capabilities of containerization, edge computing, and cloud technologies (Scolati et al., 2020). BriskStream framework enhances the performance of data stream processing by utilizing the powerful capabilities of shared-memory multicore architectures (Zhang et al., 2019). Besides of data, the availability of computing resources also changes over time, requiring stream computing systems to support continuous processing with low latency and high stability.

Several stream computing platforms, such as Cao et al. (2020), Flink (2024), Samza (2024), Spark (2024) and Storm (2024), are widely used. Among these, Apache Storm stands out due to its strong scalability and simple architecture design. However, Storm's optimization technology lacks consideration of multiple factors that influence

* Corresponding author.
E-mail addresses: sundaweicn@cugb.edu.cn (D. Sun), chenhaiyang@cugb.edu.cn (H. Chen), shang.gao@deakin.edu.au (S. Gao), rbuyya@unimelb.edu.au (R. Buyya).

https://doi.org/10.1016/j.eswa.2024.124346
Received 17 March 2024; Received in revised form 11 May 2024; Accepted 27 May 2024
Available online 1 June 2024

system performance at different stages. Therefore, there is room for improvement. For example, Storm's performance may not meet the demands of emerging scenarios, particularly when dealing with real-time input rate fluctuations during operation (Sahni & Vidyarthi, 2021). Manually configuring running parameters, or using immutable resource configurations can be infeasible or inefficient. These approaches often usually underutilize cluster resources (Liu & Buyya, 2020), leading to unstable system operation, increased latency and reduced resource utilization.

When dealing with fluctuating data rates, it is effective to adjust the system at run time for persistent performance (Röger & Mayer, 2019). However, the factors within the stream computing system are not independent of each other, and optimizing solely based on one factor may yield limited improvements, or even ineffective results due to constraints imposed by other factors.

1.1. Key contributions

In this paper, we investigate these challenges and propose a solution. We propose a framework named Sgp-Stream by orchestrating scheduling, grouping and parallelism. Sgp-Stream orchestrates **s**cheduling, **g**rouping of data streams, and operator **p**arallelism (Sgp). Within our framework, Apache Storm can rapidly adapt to fluctuating input rates and improve its performance, compared to its native strategies. Our main contributions are as follows:

- We investigate how various factors at different stages of stream application execution can influence the performance of a stream computing system. Through experiments conducted with on Apache Storm, we evaluate the impact of three distinct factors on system performance. Our results show that individual factors can only fine-tune system performance up to a certain limit. Effective performance tuning requires the coordination of these factors to meet requirements for latency, throughput and resource utilization.
- We introduce formal models for stream application represented by directed acyclic graph (DAG), multi-dimensional featured data streams, data center resource, and latency & throughput performance from a quantitative perspective. Our proposal, the Sgp-Stream framework, incorporates strategies for runtime-aware data stream grouping based on smooth weighted polling, elastic adaptive scheduling based on Linear Deterministic Greedy (LDG), and elastic scaling based on Gradient Descent (GD). The framework supports adaptive coordination at three levels, allowing these factors to quickly adapt to real-time input changes.
- We implement and integrate Sgp-Stream into Apache Storm, and evaluate its performance using large-scale Twitter data set and classic topology that handle real-time data streams. We compare Sgp-Stream with existing state-of-the-art works based on key system metrics, including application latency, average throughput and resource utilization.

1.2. Paper organization

The rest of the paper is organized as follows: Section 3 discusses the challenges and opportunities brought by Apache Storm's strategies for data stream grouping, elastic scaling and scheduling. Section 4 defines the models for stream application, data stream, resources and performance. Section 5 introduces the Sgp-Stream framework and its multi-level adaptive algorithms, including strategies for runtime-aware data stream grouping based on smooth weighted polling, elastic adaptive scheduling based on Linear Deterministic Greedy (LDG), and elastic scaling based on Gradient Descent (GD). Section 6 discusses the experimental settings and performance evaluation. Section 2 reviews state-of-the-art work relevant to task scheduling, data stream grouping and elastic scaling. Finally, conclusion and future work are given in Section 7.

2. Related work

In this section, we review three broad categories of related works: DAG scheduling strategies, stream grouping and elastic scaling strategies, as well as comparing our work with the closely related papers as shown in Table 1.

2.1. DAG scheduling strategies

Task scheduling is the most important part for a stream computing system to deploy applications. Scheduling is also one of the key factors for stream processing performance. It involves the efficient allocation of tasks under limited computational resources to minimize processing latency and ensure the continuity and stability of data processing (Nicoleta Tantalaki & Roumeliotis, 2020). It steps in after the input stream is partitioned and allocated to each task, meanwhile, the SLA service level agreement predefined by users should be satisfied (Govindarajan et al., 2017). There are different scheduling methods that dynamically map tasks onto the cluster nodes. For example, FineStream facilitates task scheduling research by decomposing computational tasks into smaller subtasks and dynamically scheduling them between CPUs and GPUs (Zhang et al., 2020). The default scheduler of Storm adopts polling for task allocation, while ignoring issues such as the communication cost between tasks, resource requirements and load balancing, etc. In recent studies, researchers tried to improve the default scheduler of Storm (Duan & Zhou, 2020; Farrokh et al., 2022; Muhammad & Aleem, 2021b; Muhammad et al., 2021).

In Muhammad and Aleem (2021a), a scheduler was proposed which considers the historical communication between tasks and the heterogeneity of computing resources. Its core target is using a resource-aware mapping mechanism for higher throughput and less application latency.

In Zhou et al. (2020), a scheduler based on topology structure was proposed, where the scheduling strategy is divided into two stages to reduce the communication overhead. A self-adjustment mechanism is applied for uniform distribution of workload between operating time points.

The scheduling of tasks needs to consider the topological structure, communication traffic and resource utilization. To better adapt to changes in data stream grouping and parallelism elastic scaling, we propose a scheduling strategy to improve the processing performance of the stream computing system.

2.2. Stream grouping

There are many data stream grouping strategies built for Apache Storm, among which, Shuffle grouping and Key grouping are the most common ones. Shuffle grouping distributes data to all the downstream tasks in a polling way. It does not store data state, so it can only process stateless data. Key grouping uses hash function to distribute data to specific downstream tasks. It ensures the data with the same key can be sent to the same task. However, the system may be overloaded when it comes to high-skew data (Zhou et al., 2019).

In Nasir et al. (2015), a partial Key Grouping (PKG) method was proposed, based on which, the "W-choices" strategy was developed to support hotkey selection of multiple tasks. However, these two methods only consider load balancing. The problem of load imbalance still exists when the application topology is large.

In Chen et al. (2017), a protocol-aware heterogeneous distributed stream computing system was proposed. It uses a "coin experiment" approach to identify potential hotkeys. This method can quickly adapt to the change of key frequency in highly dynamic data streams.

In Chen et al. (2018), a network aware grouping framework was proposed, in which each network channel is assigned a different number of tuples by weight and priority. The weight of network channel is dynamically adjusted by dynamic weight control. This method can

Table 1
Comparison of Sgp-Stream and related work.

Parameters	Related work					Sgp-Stream
	Chen et al. (2018)	Cardellini et al. (2016)	Sahni and Vidyarthi (2021)	Muhammad and Aleem (2021a)	Zhou et al. (2020)	
Performance modeling	✓	✓	✓	✓	✓	✓
Adaptive grouping strategy	✓	×	×	×	×	✓
Operator parallelism	×	✓	✓	×	×	✓
Scheduling	×	✓	✓	✓	✓	✓
Resource utilization rate	×	×	✓	✓	✓	✓

effectively deal with data skew and heterogeneous cluster environment and rationally allocate data components to different network channels.

All the above strategies have advantages, but they did not improve the grouping strategy, or consider the load change issue after the elastic scaling of downstream operators. In order to adapt each task to changes in input load and processing resources in the stream environment, the number of data stream tuples distributed needs to be dynamically adjusted for load balancing.

In this paper, the grouping strategy of data stream is improved compared to the Shuffle grouping. An adaptive grouping strategy based on smooth weighted polling is adopted to load balance downstream operators. This strategy can cooperate well with the elastic scaling and the scheduling strategies for better performance.

2.3. Elastic scaling strategies

To make the system meet the SLA requirements during operation and to ensure the low latency and high stability, different optimization methods are put forward at different levels to support elasticity (Borkowski et al., 2018; Dias de Assunção et al., 2018; Marangozova-Martin et al., 2019).

Aiming to address the issues brought by unpredictable input rate, paper (Cardellini et al., 2016) proposed two new strategies, i.e., elastic scaling and state transition for Apache Storm. The new strategies can dynamically adapt to unpredictable rates. Stream parallel compression algorithms decompose a stream compression procedure into multiple fine-grained tasks for more efficient processing. Their competing mechanisms (Zeng & Zhang, 2023) have a reference value for operator parallelism research. Among them, the elastic scaling strategy uses a simple addition plus multiplication minus method to adjust the degree of parallelism.

In Xie et al. (2017), experiments verified that a reasonable number of topology workers helped improve the system performance. The paper also proposed an algorithm that computes the resources required by the topology and adaptively allocates cluster resources. The performance comparison test of DefaultScheduler in Apache Storm proves that the algorithm improves the performance. However, this method can only analyze the parallelism of topology statically, but not support dynamic adaption to fluctuating rates at runtime.

In Sahni and Vidyarthi (2021), a new elastic scaling method was proposed from two dimensions: parallelism and resource. This method not only supports the adjustment to the parallelism of operators, but also expands the available resources on virtual machine. It can effectively adapt to the changes of data stream and improve the resource utilization.

When the parallelism of components in stream applications cannot adapt to the current streaming environment in real time, the effect of optimizing system performance in other aspects cannot reach the best possible outcome. All the above elastic scaling strategies have certain advantages, but the combination of the elastic scaling strategy with the operator parallelism and the grouping & scheduling strategies is rarely considered.

In this paper, we consider both, i.e., the elastic scaling strategy with the operator parallelism. It not only enables Sgp-Stream to adapt to the changing data stream dynamically, but also assists it in cooperating with the grouping and scheduling strategies at multiple levels. The gradient descent based elastic scaling strategy reduces the operation bottlenecks, thereby reducing the application latency and increasing the resource utilization.

3. Observations

In a big data stream computing environment, users need to manually write the business logic of their stream application and submit it to the system for analysis and deployment. The execution process of the application can be divided into three phases. In the **logical construction** phase, the user is responsible for coding the data stream application and submitting it to the stream computing platform. In big data stream computing environments, each application is commonly represented as a set of sub-vertices interconnected via data dependencies, and described by a corresponding directed acyclic graph (DAG). In the **operator parallelism and grouping** phase, the platform parses the execution logic of the DAG. In this phase, the operator parallelism settings, grouping strategy and scheduling strategy are determined. Each DAG is coordinated by multiple workers distributed across multiple worker nodes. There are multiple executors in each worker, each executor is a thread running Spout or Bolt processing logic. In the **task scheduling and computation** phase, the platform assigns the instantiated computation tasks (the basic unit of execution, instantiated from spout or bolt) to clusters for execution based on the selected scheduling strategy, while the transmission path of data stream is determined by the grouping strategy. The whole process is shown in Fig. 1.

Each phase has an influence on the system performance (Sarathchandra et al., 2021; Yudong et al., 2020). For example, in Apache Storm, users must design the DAG and configure parameters in advance. Due to the lack of knowledge or experience, these parameters might not be set optimally. It is also challenging to manually adjust key parameters (e.g., operator parallelism, task scheduling and data stream grouping strategies) in real time based on changing application states and resource consumption (Vogel et al., 2021). In cloud computing, resource elasticity allows for an application to scale out/in according to demand, which brings new ways to adjust system performance (Dias de Assunção et al., 2018). Multiple factors within a stream computing system are interconnected, and optimizing performance solely from the perspective of one factor may result in limited improvements. Considering multiple factors at various stages can improve processing bottlenecks and low resource utilization. It can also optimize the problem of resource overutilization caused by high operational loads on machine resources. Resource overutilization refers to the state where machine resources are overloaded due to high operational loads. With virtual machine CPU overcommitment technology, machines may overload resources within time slice periods. This can result in resource contention between virtual machines and may even lead to machine downtime. Therefore, excessively high resource utilization can significantly impact system performance. To mitigate processing bottlenecks and resource utilization problems, we aim to consider multiple factors at different stream processing stages (De Matteis & Mencagli, 2017; Ni et al., 2020).

In a streaming computing system, a streaming application needs to go through the coordination of multiple phases during runtime. Scheduling strategy, grouping strategy and parallelism scaling strategy are the three key aspects for optimizing system performance. Before proposing any solution, we first identify the factors influencing the system performance in each phase through case studies.

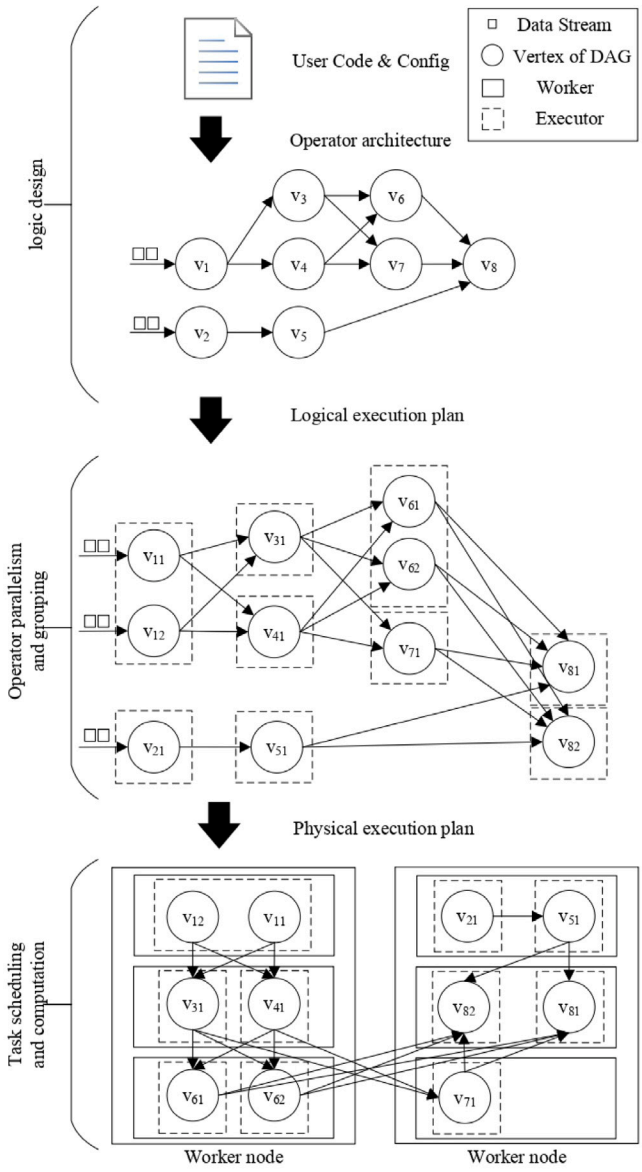


Fig. 1. Execution process of a stream application.

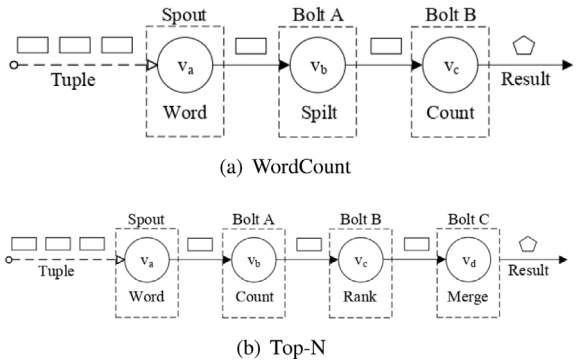


Fig. 2. Logical graphs of WordCount and Top-N.

3.1. CASE 1: DAG scheduling

In Apache Storm, a stream application is represented by a directed acyclic graph (DAG) and described as a topology. A topology is a network of spouts and bolts. A spout is a source of streams in a computation, while a bolt processes any number of input streams and produces any number of new output streams.

After the topology is submitted, Storm instantiates components (vertices in DAG) according to the application's parameter configuration and schedules tasks to run on cluster nodes through the scheduler. Different scheduling strategies influence the performance differently (Fu et al., 2015; Qureshi et al., 2020; Souravlas et al., 2021; Storm, 2024). Since the stream environment changes over time, continuous adjustments to the scheduling strategy of the stream computing system are necessary. However, the parallelism of topology cannot be dynamically adjusted by scheduling strategies alone, and the performance improvements brought by adjustments to scheduling strategies are limited (Govindarajan et al., 2017).

We run observation experiments on an Apache Storm cluster to investigate the impact of different factors on system performance while keeping the operator parallelism fixed. In these experiments, we purposely used low-performance machines for two main reasons: (1). Using low-performance machines helps us quickly identify the bottleneck factors that affect system performance without the performance pattern being influenced. (2). Compared to using high-performance machines, finding more efficient ways to utilize low-performance ones is more practical for resolving the performance issues in distributed stream computing systems.

The Top-N and WordCount topologies are the fundamental and widely used streaming applications, commonly adopted in various studies for comparisons. To ensure the universality and comprehensiveness of our experimental comparison, we have selected Top-N and WordCount topologies as benchmarks in our experiments.

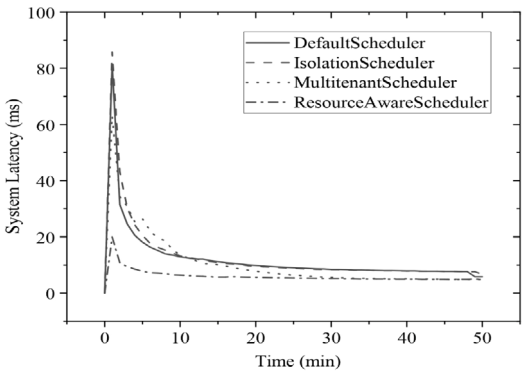
WordCount counts the frequency of words in English text. In its topology (Fig. 2(a)), the Spout vertex simulates a data stream by randomly sending sentences; Bolt A and Bolt B split sentences and calculate statistics, respectively. The Top-N topology sorts the frequency of words at given time intervals. In its topology (Fig. 2(b)), the spout vertex simulates a data stream by randomly sending words; Bolt A, Bolt B and Bolt C calculate the occurrence, sort frequency and sum results, respectively.

When a streaming computing system is in operation, each component in a streaming application is instantiated as multiple tasks executed in parallel. To thoroughly evaluate the impact of parallelism on system performance, we build the application with a wide range of parallelism degrees for each Spout or Bolt. This results in a more intricate data transmission process as data stream through multiple component tasks during operation.

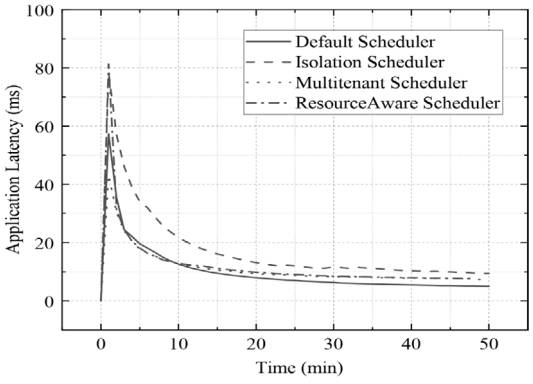
For WordCount, we keep the data input rate fixed, set the parallelism to 3 for Spout, to 1 for Bolt A and Bolt B, and use Storm's built-in schedulers DefaultScheduler, Isolation, Multitenant and ResourceAware Scheduler (Peng et al., 2015; Storm, 2024). The variation in application latency is shown in Fig. 3(a), and the relationships among system throughput, schedulers and Bolt A's capacity are depicted in Fig. 4(a). As can be seen in Fig. 3(a), the application latency stays between 6~8 ms after the system stabilizes. Under normal circumstances, the capacity (%) falls within the [0.0, 0.1] range. When the capacity value approaches 1, it indicates that the bolt is more heavily loaded and requires higher parallelism.

Through several adjustments to the input rate during the observation experiment, it was determined that a capacity within [0.0, 0.4] is optimal given the hardware conditions of the virtual machine running the bolt. This capacity ensures the stable operation of the system and facilitates the experimental observation.

In Fig. 4(a), Bolt A's capacity remains within [0.25, 0.3] under the fixed input rate. This suggests that different schedulers in this parallelism configuration are not effective in improving throughput and reduce latency. Even the ResourceAware Scheduler, which considers

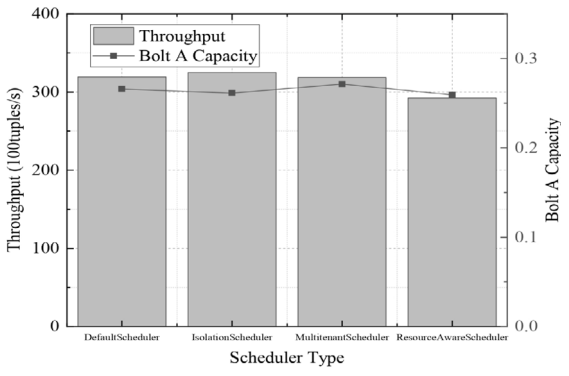


(a) WordCount

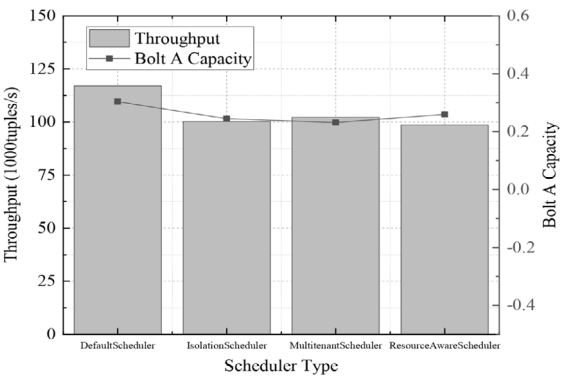


(b) Top-N

Fig. 3. Impact of different schedulers on application latency with fixed parallelism and fixed input rate.



(a) WordCount



(b) Top-N

Fig. 4. Impact of different schedulers on system throughput and Bolt A's capacity with fixed parallelism and fixed input rate.

computing resources and communication distance, does not significantly improve the system performance. The underlying reason might be that the inappropriate parallelism of Bolt vertices leads to low parallel execution ability of tasks, and the system performance cannot be substantially improved by adjusting the scheduling strategy alone.

To eliminate potential interference from topology, we also use Top-N (Fig. 2(b)) in the experiments while maintaining a fixed data input rate, setting the parallelism to 3 for Spout and 1 for the rest, and using the same schedulers. Fig. 3(b) illustrates the application latency over time and Fig. 4(b) shows Bolt A's capacity and throughput under different schedulers. The trends align with the observations made in the WordCount experiments.

To sum up, when the Storm system schedules an application, different scheduling strategies influence different aspects of performance, such as resource utilization, communication costs, application latency, and throughput. The choice of a scheduler significantly influences the system performance optimization (Liu et al., 2019). It is understandable that static configuration for a topology, set by inexperienced users, might not be optimal in a changing environment. If the parallelism of topological components is modified, the current balanced scheduling scheme may be no long appropriate, causing higher communication latency between tasks.

However, when dynamic adjustments are necessary for parallelism and the resource demands of components change, the previously balanced scheduling might also become unbalanced. This imbalance can lead to task computation latency and excessively high communication latency between tasks. Unfortunately, the built-in scheduler in the Storm system lacks the flexibility and self-adaptiveness to adjust according to the changes in parallelism. This observation has driven us to design an adaptive scheduling strategy that allows the system to adapt to dynamic adjustments in topological parallelism.

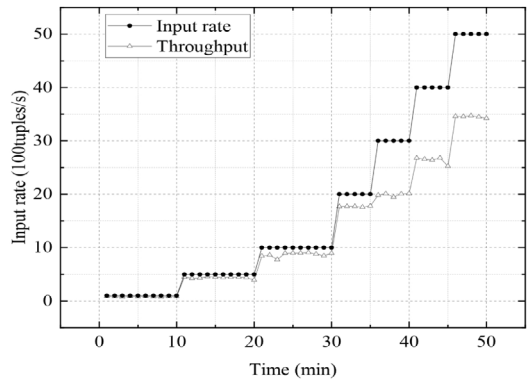
3.2. CASE 2: Operator parallelism

In this case, we investigate the performance of static topological structures under fluctuating input rates. We explore the relationship between operator parallelism and system performance, discussing the challenges related to adaptive elastic scaling strategies (Fu et al., 2017; Kombi et al., 2017).

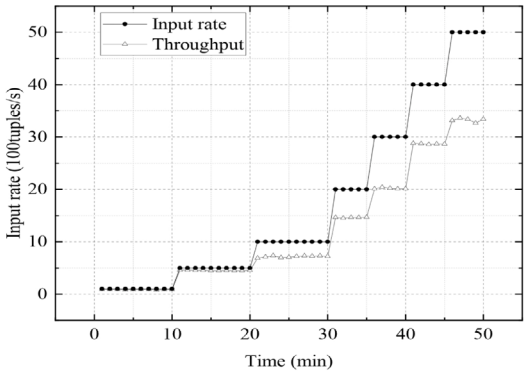
System performance metrics used in the experiment include throughput, application latency and CPU utilization (i.e. the overall CPU utilization rate of the cluster). These values can be obtained via the Storm UI or the 'top' command.

For WordCount (Fig. 2(a)), we fix the parallelism to 3 for the Spout and 1 for the remaining Bolts. Between 0 and 30 min, as we increase the input rate from 100 tuples/s to 1000 tuples/s, the system throughput roughly matches the input rate. After 30 min, as the input rate varies from 1000 tuples/s to 5000 tuples/s with an increment of 1000 tuples/s every 5 min, the system throughput does not increase proportionately. Fig. 5(a) shows the variation of input rate and throughput. When the input rate gradually increases beyond the capacity of the fixed parallelism, the throughput no longer increases proportionally to the input rate. Instead, data starts to queue in the input. To eliminate the potential interference of topology, we also use Top-N (Fig. 2(b)) with parallelism fixed to 5 for the Spout and 1 for the remaining Bolts. Fig. 5(b) shows the variation of input rate and throughput, which is consistent with the trends observed in WordCount.

Fig. 6(a) shows the changes of application latency and Bolt A's capacity in WordCount when the input rate increases from 10,000 tuples/s to 50,000 tuples/s. With the increase of input rate, the latency increases, so does the capacity of Bolt A. At the same time, the fixed parallelism configuration begins to impact the system performance. As Bolt A is reaching its processing limit, its processing rate gradually

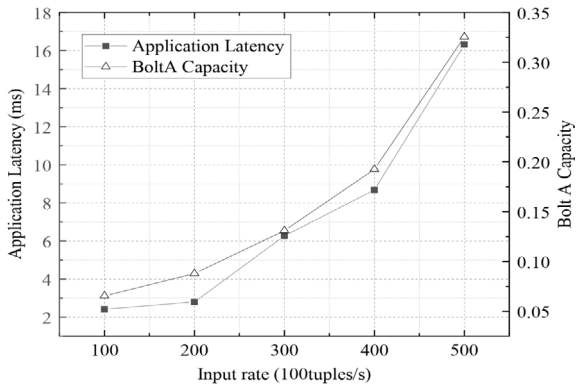


(a) WordCount

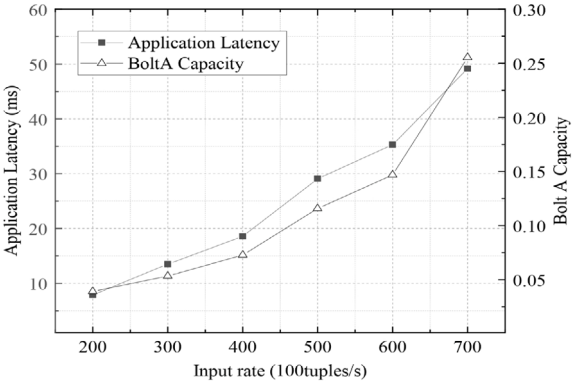


(b) Top-N

Fig. 5. Throughput in WordCount and Top-N with fixed parallelism and changing input rate.



(a) WordCount



(b) Top-N

Fig. 6. Application latency and Bolt A's capacity in WordCount and Top-N with fixed parallelism and changing input rate.

decreases and the application latency gradually increases. Fig. 6(b) shows a similar trend in Top-N as in WordCount when the input rate increases from 20,000 tuples/s to 70,000 tuples/s.

To investigate the influence of parallelism on system performance and CPU utilization, we fix the Spout parallelism of WordCount and keep a constant input rate. As shown in Fig. 7(a), at first, Bolt A's capacity stays in the range of [0.27,0.3]. When we gradually increase the degree of parallelism (executor number), both Bolt A's capacity and application latency decrease, suggesting that higher parallelism is beneficial for performance. However, when Bolt A's capacity reaches[0.01,0.1], further increases in parallelism do not lead to lower Bolt A's capacity or reduced application latency (see Fig. 7(a)), and there is no significant increase in throughput either (see Fig. 8(a)). Resource competition between component's tasks may be the reason behind this.

Meanwhile, CPU utilization decreases (see Fig. 8(a)). This decrease might be caused by task overscheduling, which increases communication costs, application latency, and lowers resource utilization. To verify these findings, we also run Top-N with fixed Spout parallelism and input rate. The results, shown in Figs. 7(b) and 8(b), follow the same trend.

The above observations reveal that there is no simple linear relationship between tuning operator parallelism and system performance, especially when the input rate fluctuates. From a quantitative point of view, increasing operator parallelism can enhance performance when the input rate exceeds the processing capability of the parallelized operators. However, after a certain point, further adjustments to parallelism yield diminishing returns, and may increase the resource and communication costs, thus negatively impacting system performance. Currently, there is a need for stream computing systems to flexibly and continuously adjust parallelism during runtime to optimize system performance in real time. How to properly adjust the parallelism for components in a topology at runtime remains a major challenge (Lombardi et al., 2018; Röger & Mayer, 2019; Wang et al., 2019).

To sum up, real-time adaptive parallelism adjustment plays an important role in improving system performance. The above observation prompts us to design an adaptive elastic scaling mechanism to fine-tune parallelism parameters. Moreover, under fluctuating inputs, system performance is closely related to parallelism configuration, load distribution, communication, and scheduling strategies. To optimize system performance, multi-level coordination is necessary.

3.3. CASE 3: Stream grouping

Stream Grouping defines how data tuples are transferred between two components that share a communication relationship and how the stream should be partitioned among the bolt's tasks. Existing research on Stream grouping strategies can be mainly categorized into two types: key based and non-key based (Son et al., 2021). The former mainly employs hash functions for stateful operators to ensure that tuples with the same hash key are mapped to the same task. The latter is more common in stream applications with stateless operators. This paper specifically focuses on non-key based grouping strategies.

Different stream grouping strategies are required for different data tuple distribution scenarios. we examine all the grouping methods supported by Storm (2024). Commonly used key-based and non-key-based grouping methods in Storm include:

- **Shuffle Grouping** randomly distributes tuples to tasks instantiated by the bolt, with each task at the same level accepting an equal number of tuples.

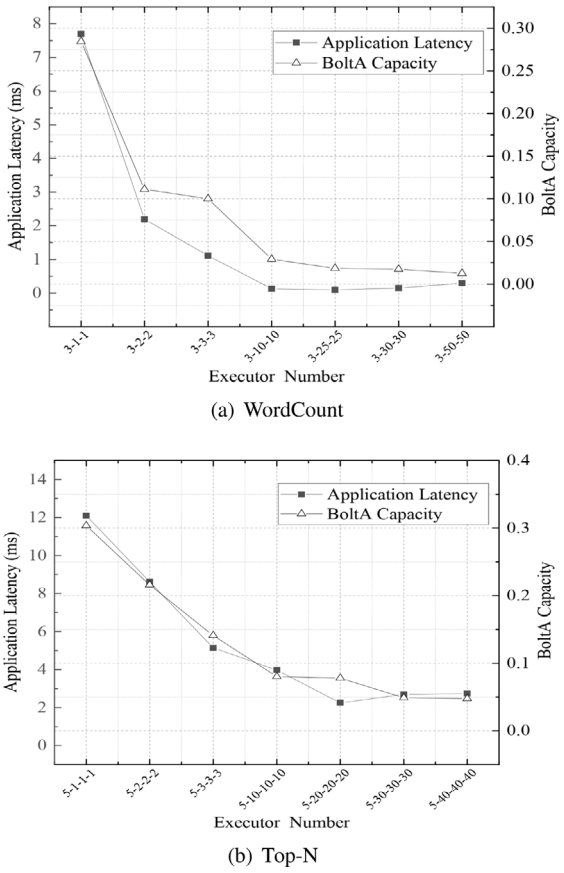


Fig. 7. Application latency and Bolt A capacity in WordCount and Top-N with fixed input rate and changing parallelism.

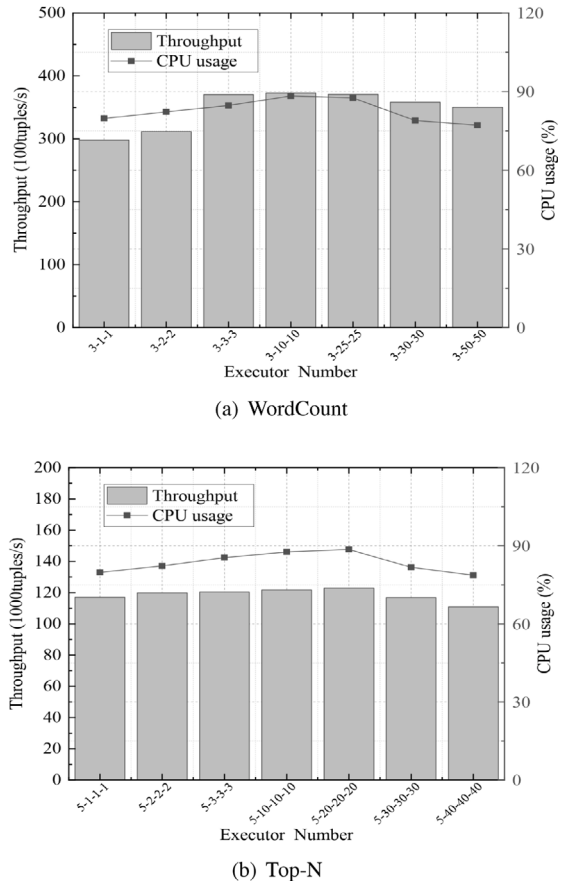


Fig. 8. Throughput and CPU usage in WordCount and Top-N with fixed input rate and changing parallelism.

- **Fields Grouping** groups tuples based on the key(s) of one or more specified fields.
- **All Grouping** copies all tuples to all tasks instantiated by the bolt.
- **Global Grouping** sends all tuples to one specific task instantiated by the bolt. Typically, Storm selects the task with the smallest task ID as the receiver.
- **Direct Grouping** directly specifies the receiving task.

In non-key-based Grouping strategies, we use the All Grouping strategy to broadcast data tuples to all tasks of the downstream operator. If a data tuple needs to be distributed to a specific task or a specified set of tasks in the downstream operator, we select the Global Grouping or Direct Grouping strategy. On the other hand, we use the Shuffle Grouping strategy when data tuples need to be randomly distributed to different tasks within the bolt. To increase the processing rate of downstream operators for tuples, we typically use the Shuffle Grouping strategy to distribute tuples in a polling mode, ensuring an even number of tuples accepted by each task of the downstream operators. To maintain consistency in message processing with a key-based grouping strategy, we usually use the Field Grouping strategy to group fields in a data tuple. Tuples with the same fields are distributed to the same task.

When the system scheduling strategy places tasks in the cluster, these tasks run on different worker nodes. However, competition for resource among workers may lead to varying resource utilization rates for worker nodes, especially in heterogeneous cluster environments. If the system uses the Shuffle grouping method, tasks with fewer resources often perform poorly, resulting in higher task and application latency and lower throughput. Obversely, tasks with more resources may not fully utilize them. The grouping method significantly affects task processing performance, which in turn impacts application latency, throughput and resource utilization (Yudong et al., 2020). This observation inspired us to design a grouping method that distributes tuples based on the real-time load of downstream operator tasks. We anticipate that this approach can improve system performance by dynamically distributing tuples to the least loaded tasks.

3.4. Challenges

Our focus in this paper is on integrating multiple layers to optimize system performance. We recognize the challenge in balancing localized optimization strategies at each level with achieving global performance through cooperative optimization across levels.

In a multi-source and dynamic stream computing environment, at the scheduling level, there exists a conflict between the communication costs among operators and the diverse, limited resources of compute nodes during runtime scheduling. We can reconcile this by scheduling complex instances onto compute nodes with lower communication costs and sufficient resources.

Regarding the elastic parallelism scaling, a tension exists between the stability of the stream computing system and the flexibility of on-demand elasticity in operators. To resolve this, we can periodically iterate to adjust the number of instances for key nodes.

At the stream grouping level, operating under vast and unbounded data streams, a discrepancy exists between the processing capacity of operators and their dynamic load states. We can address this by grouping data streams based on the load status of operators, thus reducing transmission latency across multiple data links.

4. Problem statement

This section introduces the models commonly used in big data stream computing systems, including DAG, data stream, resource and optimization objectives.

4.1. DAG model

The logic of a stream application is typically described using a directed acyclic graph (DAG) (Cugola & Margara, 2012). It is composed of a vertex set and a directed edge set, denoted as $G = (V(G), E(G))$, where $V(G) = \{v_1, v_2, \ldots, v_n\}$ represents a finite set with n vertices. $E(G) = \{e_{1,2}, \ldots, e_{i,j}, \ldots, e_{n-i,n}\} \subset V(G) \times V(G)$ is a finite set of directed edges. These edges suggest execution precedence among vertices.

A vertex represents a component that processes data stream. Each vertex v_i can instantiate one or more tasks on demand, i.e. $task_{v_i} = \{task_{v_{i1}}, task_{v_{i2}}, \ldots, task_{v_{ik}}, \ldots, task_{v_{im}}\}$, $m \in (1, 2, \ldots)$. Each instantiated task $task_{v_{ik}}$ performs the same function as defined by vertex v_i and can be scheduled onto a compute node by the scheduler.

The function of DAG is achieved by all n vertices, represented as $O = F(I)$, where I, F and O are the input data steam, function of application DAG and output data steam, respectively. In the DAG, if v_i does not have input data stream i_{v_i}, it is an input vertex, represented as v_{in}; if v_i does not have output data stream o_{v_i}, it is an output vertex, represented as v_{out}. For v_i, assume its input rate is Ir_{v_i}, the processing rate is Pr_{v_i}, the time consumed by vertex v_i running in a cluster is Tc_{v_i}, then Tc_{v_i} is related to Ir_{v_i} and Pr_{v_i}. To lower the influence of input fluctuation, we use the mathematical expectation $E_{Ir_{v_i}}$ to represent the input rate Ir_{v_i} of v_i per unit time. Tc_{v_i}, as the time consumed by vertex v_i, can be described by Eq. (1).

$$Tc_{v_i} = \frac{E_{Ir_{v_i}} \cdot \triangle t}{Pr_{v_i}}, \tag{1}$$

where $\triangle t$ represents the period from the beginning of counting the number of tuples to the end. The average processing rate Pr_{v_i} of vertex v_i can be calculated by Eq. (2).

$$Pr_{v_i} = N(Ex(v_i)) \cdot \frac{1}{m} \sum_{k=1}^{m} Pr_{task_{ik}}, m \in \{1, 2, 3, \ldots\}. \tag{2}$$

where $Pr_{task_{v_{ik}}}$ is the real processing rate of $task_{v_{ik}}$, and $N(Ex(v_i))$ is the parallelism of vertex v_i.

It can be seen that when the parallelism $N(Ex(v_i))$ increases, the processing rate $Pr_{task_{v_{ik}}}$ of vertex v_i increases, as well as the average processing rate Pr_{v_i}. Therefore, with a high data input rate Ir_{v_i} or the mathematical expectation $E_{Ir_{v_i}}$, increasing the average processing rate Pr_{v_i} can improve the average processing rate Pr_{v_i} of v_i, thus reducing the processing time Tc_{v_i}.

In a DAG, each vertex processes tuples received before sending them to downstream vertices. A directed edge between two vertices represents a transmission path from one vertex to another. If $\exists e_{i,j} \in E(G)$, then $v_i, v_j \in V(G)$, $v_i \neq v_j$, and $\langle v_i, v_j \rangle$ is an ordered pair. This ordered pair signifies a data dependency relationship between v_i and v_j, i.e., data streams from v_i to v_j. Vertex v_j cannot commence processing until the tuples have been processed by tasks of v_i.

We define $Ts_{e_{i,j}}$ as the tuple transmission time on directed edge $e_{i,j}$. It is determined by the network link bandwidth $Band_{e_{i,j}}$ between v_i and v_j (bps) and the size of output data d_{v_i} by v_i (bits). If v_i and v_j are on the same worker, their transmission time is generally considered as 0. $Ts_{e_{i,j}}$ can be described by Eq. (3).

$$Ts_{e_{i,j}} = \begin{cases} 0, & if\ v_i\ v_j\ on\ the\ same\ worker, \\ \frac{d_{v_i}}{Band_{e_{i,j}}}, & otherwise. \end{cases} \tag{3}$$

4.2. Data stream model

In a running process, data flows from upstream to downstream tasks. This continuous and unbounded sequence of data tuples is abstracted as a data stream. The ith data tuple, denoted as dt_i, which is transmitted between communication groups, can be represented by a tuple id id_i, key k_i, its value val_i and the timestamp ts_i, i.e., $dt_i = (id_i, k_i, val_i, ts_i)$. The data stream $S(G)$, which consists of multiple tuples from G, is represented by Eq. (4).

$$S(G) = \{(id_1, k_1, val_1, ts_1), \ldots (id_i, k_i, val_i, ts_i), \ldots\}. \tag{4}$$

Data tuples are distributed from upstream to downstream tasks based on the grouping strategy. To minimize data loss, a sliding window can be used to enhance reliable transmission. For stateless vertices, the sliding window can cache tuples awaiting for processing. For stateful vertices, the sliding window effectively maintains processed tuple information and saves node states.

4.3. Resource model

We represent a network data center with N compute nodes as $DC = \{cn_1, cn_2, \ldots, cn_N\}$. They are virtual machines running on physical computers.

Resources of compute node cn_i can be measured in various dimensions, such as CPU, memory, network bandwidth and I/O. We use $Pw(cn_i)$ to denote the computing power of node cn_i. It can be calculated by considering the CPU resources and system memory resources of cn_i in a weighted manner, as described by Eq. (5).

$$Pw(cn_i) = \alpha \cdot CPU_{cn_i} + (1 - \alpha) \cdot Mem_{cn_i}. \tag{5}$$

Here, the resource weight coefficient α is a constant determined based on the ratio of CPU-type to Memory-type components in the committed topology, where Mem_{cn_i} represents the compute node cn_i's memory resources. CPU_{cn_i} is cn_i's Floating Point Operations Per Second ($flops$), and is described by Eq. (6).

$$CPU_{cn_i} = Core \cdot \frac{Cycles}{Second} \cdot \frac{flops}{Cycle}, \tag{6}$$

where $Core$ indicates the total number of cores a processor has, $\frac{Cycles}{Second}$ is the clock frequency (Hz) of a core, and $\frac{flops}{Cycle}$ is the total number of floating-point operations per cycle. Floating-point operations per second (FLOPS, flops, or flop/s) is a measure of computer performance that is useful in scientific computing scenarios where floating-point calculations are required.

4.4. Optimization objectives

Low latency of stream application and high throughput are two critical performance requirements for stream computing system (Karimov et al., 2018). In this section, we profile mathematical relationships between the application latency, throughput and data streams. We also outline the constraints required to achieve low application latency and high throughput in common stream computing environments.

Within these systems, data tuples are transmitted in multiple streams among vertices. If the processing of one tuple fails, it is considered a $failed$ data tuple; otherwise, it is an $Acked$ data tuple. The ACK mechanism ensures that data tuples are processed properly. The $Acked$ tuples contribute to the application's throughput. We refer to the amount of data properly processed per unit time of the system as the system throughput. We represent system throughput as $T_p(G)$. Higher throughput indicates better performance.

The application latency consists of two components: computation latency and transmission latency. The application latency is deemed acceptable by users if it remains at the millisecond level. We use $T_l(G)$ to denote the application latency. It includes the input tuple's transmission time from the input to the output vertex and the processing time

of each vertex. For the convenience of calculation and without loss of generality, we assume the DAG has only one input vertex v_{in}, and one output vertex v_{out}.

We denote the directed path from vertex v_i to v_j as $P_{v_i,v_j} = \langle v_i, \ldots, v_j \rangle$, and the application latency $T_l\left(p_{v_i,v_j}\right)$ of one tuple can be calculated by adding up all the vertices' processing time and the data tuple' transmission time along the path P_{v_i,v_j}. It can be described by Eq. (7):

$$T_l\left(p_{v_i,v_j}\right) = \sum_{v_k \in p_{v_i,v_j}} Tc_{v_k} + \sum_{v_i,v_k \in p_{v_i,v_j}} Ts_{e_{v_i,v_k}}. \tag{7}$$

Here, $Ts_{e_{v_i,v_k}}$ represents the transmission time of one data tuple on the directed edge e_{v_i,v_k} between a pair of communication vertices v_i and v_k, and Tc_{v_k} represents the tuple processing time spent by vertex v_k.

The application latency of the DAG for one tuple is the maximum time in the directed path from input vertex v_{in} to output vertex v_{out} when processing that tuple. It can be described by Eq. (8).

$$T_l(G) = \max_{p_{v_{in},v_{out}} \in p_{v_{in},v_{out}}(G)} \left(T_l\left(p_{v_{in},v_{out}}\right)\right), \tag{8}$$

where, $p_{v_{in},v_{out}}(G)$ is the set of directed paths in G, starting from the input vertex v_{in} and ending at the output vertex v_{out}. The critical nodes v_i are determined based on the method where the earliest time $EST(v_i)$ and the latest time $LST(v_i)$ of the data stream arriving at the vertex are equal. The computation latency and transmission latency are used as the weights of the activities according to the critical path method (Sun & Huang, 2016). The vertex path with the maximum latency between v_{in} and v_{out} is called a critical path $G_{v_{in},v_{out}}$ of a tuple, and all nodes on the critical path are critical nodes. Under fluctuating input, different tuples may have different critical paths. For each tuple, the critical path is always the one with the longest latency. The focus is on the critical paths of G as a whole, rather than the individual path for a specific tuple.

The Sgp-Stream framework aims to optimize several aspects: (1) Data scheduling by considering resource and communication cost and adopting an elastic scheduling strategy. (2) Data grouping by sensing the downstream load states. (3) Operator parallelism by tuning the parallelism degree to adapt to the input rate. (4) Coordination of factors from multiple dimensions in real-time to enable continuous system performance optimization.

5. Sgp-Stream overview

Our Sgp-Stream framework addresses the unique challenges posed by multi-stage data processing in data stream computing. These challenges involve the adjustment of scheduling, grouping, and parallelism degree, which are not inherently cooperative and are often constrained by the architecture of stream computing systems.

In this section, the overall structure of Sgp-Stream is discussed, including its system architecture, online monitoring, and algorithms for adaptive grouping based on workload, elastic scaling based on heuristic techniques, and lightweight scheduling.

5.1. System architecture

Built on top of Apache Storm, Sgp-Stream consists of several key subsystems: Nimbus, Zookeeper, and Supervisors, as shown in Fig. 9. Nimbus subsystem serves as the primary node in the cluster, responsible for tasks like receiving code from users, creating DAG diagrams based on the code logic, and deploying these diagrams to the appropriate Supervisors. Additionally, Nimbus plays a role in monitoring and coordinating the work of Supervisors and Nimbus itself, swiftly restarting any failed workers in Supervisor subsystem. Various scheduling strategies can be implemented through the IScheduler interface. Storm 1.1.0 or later offers support for 4 built-in schedulers: Default Scheduler, Isolation Scheduler, Multitenant Scheduler, and Resource-Aware Scheduler. Users can further customize the scheduling strategy in the storm.yaml configuration file.

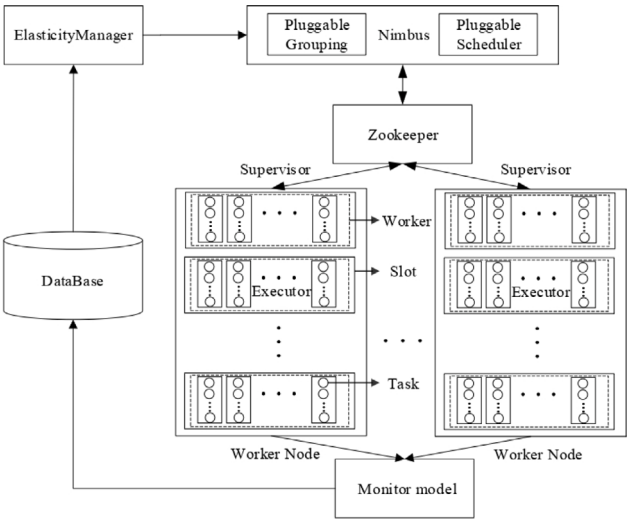


Fig. 9. Sgp-Stream architecture.

Zookeeper subsystem function as a centralized service for maintaining configuration information. It communicates with Nimbus and Supervisors, preserving information about the working status of Nimbus and Supervisors, and ensuring proper coordination across the cluster. Each Supervisor subsystem is responsible for managing the worker processes, enabling a number of worker slots based on the hardware capacity of the machine it runs on, and overseeing the execution of tasks.

Sgp-Stream supports dynamic elasticity in real time, with 3 essential steps:

(1) It provides dynamic adaptive scheduling via the IScheduler interface. DAG graphs are initially scheduled based on the available resources. They are then incrementally adjusted by the elastic scaling strategy. (2) An online monitoring module is deployed in each Supervisor subsystem. This module stores all collected performance statistical information in a database, including data stream transmission rates, task processing rates, and node performance information. This data is used to monitor the load of downstream operator tasks and the performance changes of compute node running tasks, helping determine if the current data stream distribution strategy remains suitable. (3) The elastic scaling module reads information from the database to identify any operators causing parallelism bottlenecks. If the elastic scaling conditions are met, the module recalculates the parameter plan of the operator parallelism and executes the "Rebalancing" command to reconfigure the system's resources.

5.2. DAG scheduling algorithm

The scheduler is responsible for arranging the execution of submitted stream applications based on specific scheduling strategies. Our primary goal is to minimize communication cost between tasks during scheduling while also balancing the load in heterogeneous clusters for performance purposes. Communication overhead is mainly influenced by 3 factors: inter-process communication between compute nodes, intra-process communication on a single compute node, and communication among threads within a process. In this paper, we focus on addressing the communication costs associated with inter-process and intra-process communication, as these are generally more expensive (Li & Zhang, 2017).

To meet these objectives, we design a resource-aware scheduling strategy based on Linear Deterministic Greedy (LDG). This strategy consists of 3 steps: (1) Calculate the appropriate number of workers according to the resource requirements of the topology and the available resources of compute nodes. (2) Allocate the executors, which are threads spawned by worker processes responsible for running tasks, with high communication loads to the same worker and ensure that these workers are placed on the same compute node. (3) Balance load based on the workers' maximum available capacity and the executors' resource requirements. To calculate the number of workers suitable for the topology, we collect runtime information such as data transmission rates between executors, resources consumed by each executor, and the availability of resources on the compute nodes. The appropriate number of workers, denoted as k, can be calculated as shown in Eq. (9).

$$k = \frac{Re(G)}{C_{avg}}, k \in \{1, 2, 3, \ldots\}, \tag{9}$$

where $Re(G)$ represents the amount of resources used by the thread executor instantiated by the Spout or Bolt in the DAG G, and C_{avg} represents the maximum capacity value set for each worker process on the compute node. To handle heterogeneity among compute nodes, we calculate the average capacity of each worker C_{avg} using Eq. (10).

$$C_{avg} = \frac{1}{N}\sum_{i=1}^{N}\frac{Pw\left(cn_i\right)}{Slot\left(cn_i\right)}, Slot_i \in \{1, 2, 3, \ldots\}. \tag{10}$$

$Slot\left(cn_i\right)$ is the number of available slots on compute node cn_i. It indicates the percentage of resources that can be used. $Pw\left(cn_i\right)$ represents the computing power of the ith node cn_i in the cluster.

To evaluate the communication cost and load when allocating an executor to a worker, we construct an objective function described by Eq. (11).

$$g\left(Executor, P_i\right) = T\left(Executor, P_i\right) \cdot L\left(P_i\right), \tag{11}$$

where $g(Executor, P_i)$ represents the construction objective function that evaluates the communication cost and load that the thread executor assigns to the worker P_i.

$T(Executor, P_i)$ represents the statistical traffic between the $Executor$ to be allocated and the already allocated executors on worker P_i. Larger values of $T(Executor, P_i)$ indicate lower communication costs. $L(P_i)$ captures the load constraint on worker P_i, as expressed in Eq. (12).

$$L\left(P_i\right) = 1 - \frac{load_{P_i}}{C_{avg}}, \tag{12}$$

where $load_{P_i}$ signifies the load generated by all executors on P_i, and C_{avg} represents the average computing resource as calculated in Eq. (10). It is important to note that the topology requires resources while the cluster is running; otherwise, it cannot be committed successfully, so the average resource must be greater than 0. To minimize inter-process communication cost and balance the load across compute nodes, we first select the worker with the highest evaluation function, that is, $Max\left(g\left(Executor, P_i\right)\right)$ for allocating executors. Then, we allocate the required k workers based on the available resources of each compute node. We follow a rule of placing workers on the same node to load balance the cluster. This scheduling strategy is described in Algorithm 1.

Algorithm 1 Scheduling Algorithm

Input: G, current available capacity $Pw(Node)$ of compute nodes in the data center DC.
Output: Vertices scheduling scheme on compute nodes.

1: $G = (V(G), E(G))$
2: $DC = \{cn_1, cn_2, \ldots, cn_N\}$
3: **if** G or number of available compute nodes is null **then**
4: Return null.
5: **end if**
6: **for** each vertex v_i in G **do**
7: Determine the state of v_i in G according to its function.
8: Determine the degree of parallelism for each vertex v_i in G based on system configuration parameters.
9: **if** v_i is a stateful vertex **then**
10: Create a new vertex v_i' for sharing the states of all tasks.
11: **end if**
12: **end for**
13: Calculate the number of workers k for the topology by Eq. (9).
14: Calculate the maximum capacity of each worker C_{avg} in the cluster by Eq. (10).
15: Create the collection $Executors$ of G, and get all executors of G and add to $Executors$.
16: **while** $Executors$ is not null **do**
17: **for** each $Executor$ in $Executors$ **do**
18: Create the unallocated set $Unassigned$, and add the pending $Executor$ to $Unassigned$.
19: **for** $i = 1$ to k **do**
20: $T(Executor, P_i)$, get communication traffic between the $Executor$ and the existing executors on worker P_i based on statistics.
21: Calculate the load constraint $L(P_i)$ of worker P_i by Eq. (12).
22: Calculate the evaluation function $g\left(Executor, P_i\right)$ by Eq. (11).
23: **end for**
24: Get $Max\left(g\left(Executor, P_i\right)\right)$, the ith worker P_i that maximizes the evaluation function.
25: **if** Assign $Executor$ to the specified worker P_i **then**
26: Remove $Executor$ from $Unassigned$.
27: **end if**
28: **end for**
29: **end while**
30: **if** $Unassigned$ is not null **then**
31: Poll and schedule unassigned $Executor$ in $Unassigned$ based on current workload of worker P_i.
32: **end if**
33: **for** $i = 1$ to k **do**
34: Sort N nodes in DC by current available capacity in descending order.
35: Assign worker P_i polled to compute nodes.
36: **end for**
37: **return** Vertices scheduling scheme on compute nodes.

In Algorithm 1, the inputs include the application DAG graph G and the set of available resources $Pw\left(cn_i\right)$ for each compute node cn_i in the cluster DC. The output is a vertices scheduling scheme on compute nodes.

Start by creating a shared state node v_i' from the stateful node in G (Step 1 to Step 12). We then calculate the evaluation function size $g\left(Executor, P_i\right)$ based on the traffic $T(Executor, P_i)$ between the executor to be allocated and the allocated executors on each worker, as well as the load constraints $L(P_i)$ for each worker P_i.

The worker with the maximum value of the evaluation function g, i.e., $Max\left(g\left(Executor, P_i\right)\right)$, is selected to allocate the executor. If there are still unallocated executors, we poll the allocation based on worker load (Step 13 to Step 32).

At the end, we determine the weight of CPU and memory according to the type of DAG diagram to calculate the available resources $Pw\left(cn_i\right)$ for each compute node cn_i in the cluster. We sort them based on computing capacity from largest to smallest, and assign the polled worker P_i to the compute nodes, ensuring that all workers in G can

quickly allocate appropriate resources and load balance the nodes in G (Step 33 to Step 36).

This scheduling strategy adjusts the location of executor and worker within the compute node. It places the executor into the worker in turn based on the communication cost and the amount of resources, minimizing executor communication costs and ensuring a more balanced utilization of worker resources.

5.3. Stream grouping algorithm

The stream grouping strategy distributes data tuples from the upstream to downstream tasks. To make each downstream task adapt to changes in input load and available resources, it is necessary to dynamically adjust the distribution of tuples for load balancing. Based on the smooth weighted grouping strategy (Nandal et al., 2021), we use a weighted polling algorithm to direct upstream tuples to lightly loaded downstream tasks. This reduces the risk of overloading the already heavily loaded tasks and improves the resource utilization rate.

Assume that v_i is an upstream vertex of v_j, i.e. $\langle v_i, v_j \rangle$ is an ordered pair. There are m tasks on v_j, denoted as $task_{v_j}$. To support the weighted polling process, we use three sets of weights (W_j, CW_j and EW_j). We initialize the configured weight $W_j = \{w_{j1}, w_{j2}, \ldots, w_{jm}\}$, $m \in (1, 2, \ldots)$ based on the load value of each task in $task_{v_j}$. Higher load indicates smaller weight value. Current weight value of $task_{v_j}$ is denoted as $CW_j = \{cw_{j1}, cw_{j2}, \ldots, cw_{jm}\}$, where $m \in (1, 2, \ldots)$. The effective weight, represented as EW_j and $EW_j = \{ew_{j1}, ew_{j2}, \ldots, ew_{jm}\}$, also depends on $m \in (1, 2, \ldots)$. At the beginning, current weight CW_j is initialized to 0 for each term, and the effective weight EW_j is aligned with the configured weight W_j, as expressed by Eq. (13).

$$EW_j = CW_j + W_j. \tag{13}$$

When v_i sends a tuple to downstream $task_{v_j}$, we allocate the tuple to the task with the highest ew_{jk}, which is determined by taking the maximum of EW_j. Subsequently, the value of the effective weight EW_j is synchronized with the current weight CW_j, and the weight CW_{jk} of the selected task is updated within the current weight CW_j. It can be expressed by Eq. (14).

$$cw_{jk} = ew_{jk} - Sum(W_j), \tag{14}$$

where parameter $Sum(W_j)$ can be calculated by Eq. (15).

$$Sum(W_j) = \sum_{k=1}^{m} w_{jk}, \tag{15}$$

$Sum(W_j)$ represents the sum of configured weights of all tasks of v_j. We update the weight values cw_{jk} in CW_j, and this updated CW_j is used to select the tuple receiver from the downstream tasks during the next iteration. The assignment iteration is considered complete when the values of all weights in the current weight CW_j are reduced to 0, as specified by Eq. (16).

$$\sum_{k=1}^{m} cw_{jk} = 0, m \in \{1, 2, 3, \ldots\}. \tag{16}$$

To dynamically perceive the new load state of each task of v_j, we adjust the weight of each task in the configured weight W_j according to their current load state. First, we analyze run-time metrics, such as data volume, input queue waiting time, and processing time. The average tuple processing rate v_{rate} measures the task load and can be calculated by Eq. (17).

$$v_{rate} = \frac{Data_t}{T_u}, \tag{17}$$

where $Data_t$ represents the number of tuples processed within the time frame of $coolTime$, T_u is the time required for processing a tuple within $coolTime$, which includes the tuple's input queue waiting time and processing time. $coolTime$ is related to the start of the elastic scaling strategy. This process allows us to adjust the weight in W_j based on the load of each task, reducing the weights for heavily loaded tasks and increasing them for those with lighter load. The Stream Grouping Algorithm is described in Algorithm 2.

Algorithm 2 Stream Grouping Algorithm

Input: task set $task_{v_j}$ of downstream vertex v_j, input tuple $tuple$, configured weight W_j, cooling-down time $coolTime$ and adjustable parameter λ.
Output: $targetTask$ to receive the $tuple$

1: $task_{v_j} = \{task_{j1}, task_{j2}, \ldots, task_{jm}\}, m \in (1, 2, \ldots)$
2: Get a collection of weights W_j for all tasks of v_j.
3: **if** W_j not exist **then**
4: Initialize weight W_j of set $task_{v_j}$.
5: **else**
6: Initialize EW_j by Eq. (13) .
7: Select the largest $ew_{jk} = Max(EW_j)$ from the effective weights EW_j for all tasks of v_j.
8: Synchronize the effective weight EW_j and the current weight CW_j for all tasks of v_j.
9: Assigns $tuple$ to $task_{jk}$ with weight cw_{jk}, $targetTask \leftarrow Assigns(cw_{jk}, tuple)$
10: Calculate $Sum(W_j)$ by Eq. (15) .
11: Calculate CW_j by Eq. (16).
12: Get the cooling-down time $coolTime$ from the configuration file.
13: Get $updateTime$ since the last weight update.
14: **if** $updateTime \geq coolTime$ **then**
15: **for** each $task_{jk} \in task_{v_j}$ **do**
16: Get statistical processing time $processTime$, the total number of tuple $Data$ and the amount of time T_u within the period of $coolTime$.
17: Calculate v_{rate} by Eq. (17) .
18: **end for**
19: Calculate average tuple execution rate $average(v_{rate})$ for all tasks in $task_{v_j}$.
20: $averageTime \leftarrow average(processTime)$
21: **for** each $task_{jk} \in task_{v_j}$ **do**
22: **if** $v_{rate}[task_{jk}] \geq average(v_{rate})$ **then**
23: $w_{jk} \leftarrow w_{jk}/2$
24: **else**
25: $w_{jk} \leftarrow w_{jk} + \lambda$
26: **end if**
27: **end for**
28: $Update(W_j)$
29: **end if**
30: **end if**
31: **return** $targetTask$ to receive the $tuple$

In Algorithm 2, the inputs include the task set of the downstream vertex v_j, denoted as $task_{v_j}$, the configured weight W_j, the cooling-down time $coolTime$, an adjustable parameter λ, and the data stream tuple $tuple$ sent from $task_{v_i}$ to $task_{v_j}$. The output of the algorithm is $targetTask$ in $task_{v_j}$, which represents the receiver of the $tuple$.

First, we obtain the weight W_j for each task of v_j. For the first run, weight W_j is set to a default value chosen by the user (Step 2 to 5). After obtaining the configured weight W_j for the tasks, we check the current load of the downstream tasks in vertex v_j. The task with the highest weight in CW_j is selected to receive the tuple first. Then, the current weight cw_{jk} of this task and the overall weight CW_j of all the other tasks are updated (Step 6 to 11).

We then analyze the average real-time tuple processing rate v_{rate} for all tasks. This analysis informs the dynamic adjustment of weight W_j. If the task's average processing rate v_{rate} is higher than the average rate

$average(v_{rate})$ for all tasks, it indicates a high-load state, and the task's weight is reduced. Otherwise, if the task has a lower load, its weight is increased. To minimize frequent weight calculations and update frequency, we introduce a cooling-down period, denoted as $coolTime$, to reduce the update frequency. This strategy enables the system to quickly adapt to changing data streams (Step 14 to 29).

In summary, the grouping strategy involves adjusting the allocation of data streams to less loaded tasks in downstream operators. The strategy allocates the adjusted data stream to the task with the lowest real-time load for processing and updates the load status of the current task. This helps reduce the processing strain on overloaded tasks, improves resource utilization, and ensures a more even distribution of tuples among downstream vertex tasks.

5.4. Elastic scaling algorithm

The elastic scaling strategy is used to identify critical nodes (vertices) in real time bottlenecks. A heuristic algorithm is employed iteratively to calculate new parallelism for each bottleneck node (vertex).

The first step is to identify the bottleneck vertices on the critical path $G_{v_{in},v_{out}}$ within G. Assuming that v_i is a critical node on this path $G_{v_{in},v_{out}}$, and m is the number of tasks of v_i. We use the set $Ex\left(v_i\right)$ to represent the n available executors for the critical node v_i, described by Eq. (18).

$$Ex\left(v_i\right) = \left\{ex_{i,1}, ex_{i,2}, \ldots, ex_{i,n}\right\}, n \in (1, 2, \ldots, m). \tag{18}$$

In this equation, $ex_{i,k}$ denotes the kth executor of v_i. $U_{ex_{i,k}}$ represents the CPU utilization of executor $ex_{i,k}$, while $U(v_i)$ signifies the average CPU usage of all executors in v_i, which can be calculated by Eq. (19).

$$U(v_i) = \frac{1}{n}\sum_{k=1}^{n}\left(U_{ex_{i,k}}\right). \tag{19}$$

If the average CPU utilization $U(v_i)$ exceeds the maximum threshold θ_{max}, it indicates the processing capability of critical node v_i cannot satisfy the current input rate and is likely to result in a long waiting queue. At this point, v_i is identified as a performance bottleneck vertex, and it becomes necessary to adjust its parallelism to improve processing capacity. On the other hand, if the average CPU utilization $U(v_i)$ is lower than the minimum threshold θ_{min}, the number of executors $N(Ex(v_i))$ needs adjustment for better resource utilization.

To adjust the parallelism of v_i, we first calculate the current bottleneck degree $f\left(v_i\right)$ based on the average input rate Ir_{v_i} and the processing rate Pr_{v_i} of v_i. Then we determine the adjustment priority for each node on the critical path based on their respective bottleneck degrees. This is described by Eq. (20).

$$f\left(v_i\right) = \frac{Pr_{v_i}}{Ir_{v_i} + Pr_{v_i}}. \tag{20}$$

The closer $f\left(v_i\right) \in (0, 1)$ approaches to 0, the higher chance that the processing rate Pr_{v_i} of v_i fails to match the input rate Ir_{v_i}. In such cases, we increase the number of executors $N(Ex(v_i))$ on the fly to lower the risk of bottleneck, and vice versa.

The value of $f\left(v_i\right)$ indicates the adjustment priority of v_i. We use Gradient Descent (GD) to iteratively calculate the parallelism, making it satisfy the current input rate. The calculation is described by Eq. (21).

$$N_{t+1}(Ex(v_i)) = N_t(Ex(v_i)) - \xi \cdot \frac{(\frac{1}{m}\sum_{k=1}^{m} Pr_{task_{ik}}) - Ir_{v_i}}{Ir_{v_i}}. \tag{21}$$

where, $N_t(Ex(v_i))$ represents the parallelism of v_i at time t. Based on current monitoring information, we can obtain each task's processing rate $Pr_{task_{ik}}$ of v_i, followed by the average input rate Ir_{v_i} and the parallelism N_{t+1} of node v_i at time $t+1$. The parameter ξ is used to adjust the speed of scaling. We set ξ to avoid elastic scaling being too slow or too fast during the system operation period. A scaling cooling time (or thread sleeping time) is also set to allow the adjustment to take effect and the monitoring module to catch the updates. The elastic scaling strategy is described in Algorithm 3.

Algorithm 3 Elastic Scaling Algorithm

Input: critical path $G_{v_{in},v_{out}}$, max threshold θ_{max}, min threshold θ_{min}, scaling cooling time $scalCoolTime$.
Output: parallelism N_t of v_i

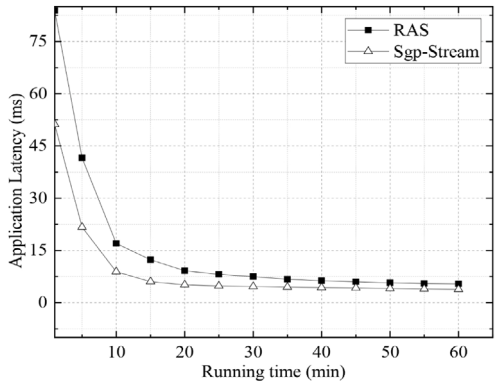
1: Get $uTime$, the time elapsed since the last parallelism N_t update.
2: **if** $uTime \geq scalCoolTime$ **then**
3: **for** each $v_i \in G_{v_{in},v_{out}}$ **do**
4: Get all the executors $Ex\left(v_i\right)$.
5: Get $N(Ex(v_i))$, the size of $Ex(v_i)$.
6: Get the CPU usage $U_{ex_{i,k}}$ of all executors $Ex\left(v_i\right)$ of v_i.
7: $Ex\left(v_i\right)$, get all executor of v_i.
8: Calculate $U(v_i) = \frac{1}{N(Ex(v_i))}\sum_{k=1}^{N(Ex(v_i))}\left(U_{ex_{i,k}}\right)$
9: **while** $U(v_i) > \theta_{max} \parallel U(v_i) < \theta_{min}$ **do**
10: Get the data input rate Ir_{v_i} of v_i
11: Get the average processing rate Pr_{v_i} of v_i
12: Calculate $f\left(v_i\right)$, the bottleneck degree of v_i by Eq. (20)
13: Calculate N_{t+1}, the parallelism of v_i by Eq. (21)
14: $Num(task_{v_i})$, get the number of all tasks of v_i
15: **if** $N_{t+1} \leq 0$ **then**
16: $N_{t+1} \leftarrow 1$
17: $Break$
18: **end if**
19: **if** $N_{t+1} \geq Num(task_{v_i})$ **then**
20: $N_{t+1} \leftarrow Num(task_{v_i})$
21: $Break$
22: **end if**
23: **end while**
24: $N_t \leftarrow N_{t+1}$
25: **end for**
26: **end if**
27: **return** N_t

In Algorithm 3, we use a heuristic algorithm to adjust the parallelism N_{v_i} of node v_i on the critical path $G_{v_{in},v_{out}}$. First of all, the average CPU usage $U(v_i)$ of critical node v_i is calculated by monitoring the CPU usage $U_{ex_{i,k}}$ of each executor in the critical node v_i (step 8). When the average CPU usage $U(v_i)$ of v_i exceeds the maximum threshold θ_{max} or is lower than the minimum threshold θ_{min}, the optimal parallelism N_{v_i} (step 9 to 13) is calculated iteratively. The threshold parameter can be set based on the historical data analysis of the monitoring module. The node's parallelism is bounded between 1 and the maximum $Num(task_{v_i})$, which corresponds to the number of tasks of v_i (step 14 to 22). To avoid potential performance fluctuations brought by frequent changes in the elastic scaling module, it is recommended to set the scaling cooling time $scalCoolTime$ properly, which can be synchronized with the "Rebalance" time (1 min by default).

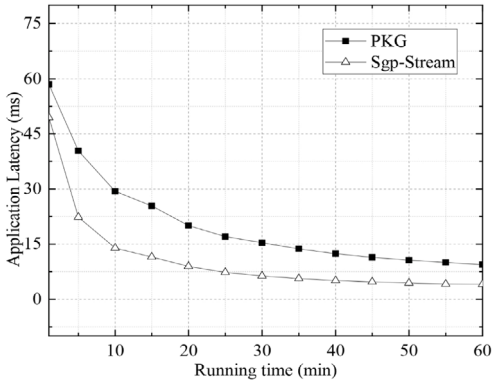
The elastic scaling strategy involves modifying the parallelism of spout/bolt in the critical path. This strategy iteratively adjusts the number of spout/bolt's executors, evaluating spout/blot bottlenecks based on CPU resource utilization. This elastic strategy can improve spout/bolt's parallel processing capabilities of spout/bolt components.

6. Performance evaluation

This section evaluates performance of Sgp-Stream and presents the experimental environment, large-scale data set, and the analysis of the results.

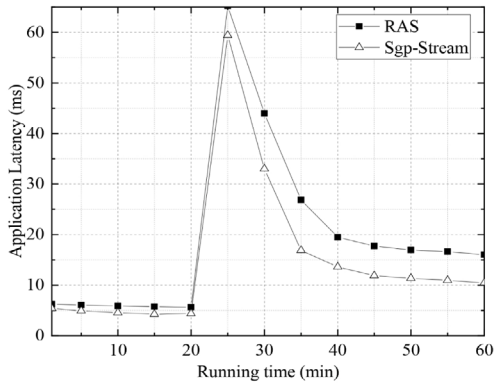


(a) RAS strategy

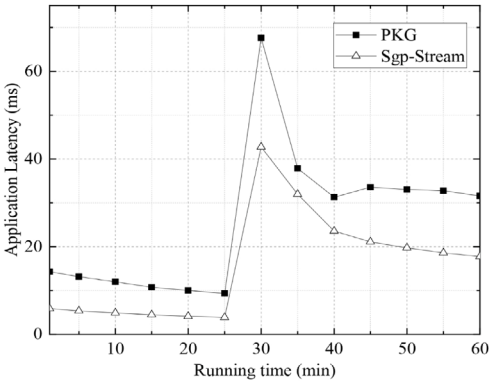


(b) PKG strategy

Fig. 10. Application latency of WordCount under a stable input rate of 6000 tuples/s.



(a) RAS strategy



(b) PKG strategy

Fig. 11. Application latency of WordCount under fluctuating input rates.

Table 2
Software configurations of Sgp-Stream.

Software	Version
OS	Ubuntu 20.04.1 64 bit
Storm	Apache-Storm-2.1.0
JDK	Jdk1.8 64 bit
Python	Python 2.7.2
Zookeeper	Zookeeper-3.4.14
Kafka	Kafka-2.3.0
Redis	Redis-6.0.5

6.1. Implementation and evaluation methodology

Apache Storm, the most popular distributed stream processing platform, is used in our experiments. It supports scalability, fault tolerance, and real-time data stream processing on clusters.

Our Sgp-Stream is developed on top of Apache storm 2.1.0 and installed on Ubuntu Server 20.04.1. It is deployed on Ali Cloud computing platform. The cluster consists of 28 machines, with one designated as master node running Storm Nimbus, two designated as Zookeeper nodes, and the rest 25 machines working as Supervisor nodes. Each machine is running Ubuntu 20.04.2 LTS with 2-Core, Intel(R) Xeon(R) CPU X5650, 2.67 GHz, 2 GB Memory, and 100M bps network interface card. All the machines in the cluster are connected. The software configuration of Sgp-Stream is shown in Table 2.

The performance of Sgp-Stream is compared the existing state-of-the-art optimization strategy in Apache Storm. Apache Storm's ResourceAware Scheduler(RAS) strategies are used. Partial Key Grouping (PKG) sets up custom stream grouping strategies.

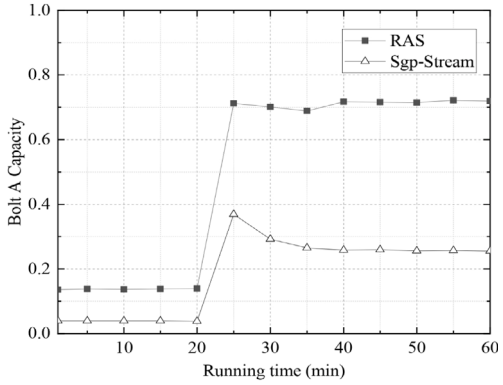
We use real-time data streams built from large-scale Twitter data set as data sources to access the WordCount topology. It is a typical data stream application that counts the frequency of words in text and consumes considerable CPU resources during execution. Different operators in this topology have different computational complexity and require different resources. Its logical structure is shown in Fig. 2(a).

To demonstrate the effective utilization of limited resources in the cluster by the Sgp-Stream framework, we intentionally set each compute node with lower hardware configurations. This enables us to observe how the framework manages the fluctuating data stream, while verifying that the Sgp-Stream framework, through its collaborative & multi-dimensional adaptive adjustment, achieves optimal system performance.

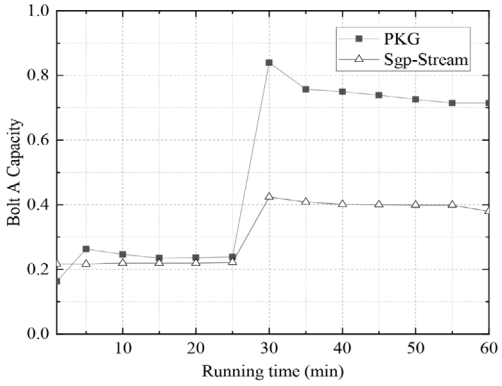
6.2. Application latency

Stream computing systems need to process data in real time, so the application latency is an important metric. It must be at the acceptable millisecond level. Usually, the lower the application latency, the higher the real-time processing capability. We use the Storm UI to detect the application latency, which is measured periodically (every 60 s) at runtime. In the experiments, we use Kafka to control the data input rate, simulating the fluctuating load in real-life scenarios.

First, we set the input rate to 6000 tuple/s. Under this rate, Sgp-Stream has better application latency than RAS. As shown in Fig. 10, when the data input rate is stable, the average application latency of Sgp-Stream and of Storm in the stable phase are about 4.927 ms and 6.699 ms, respectively. Under an input rate of 6000 tuple/s, Sgp-Stream

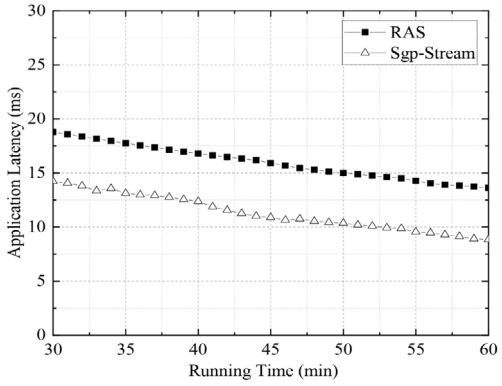


(a) RAS strategy

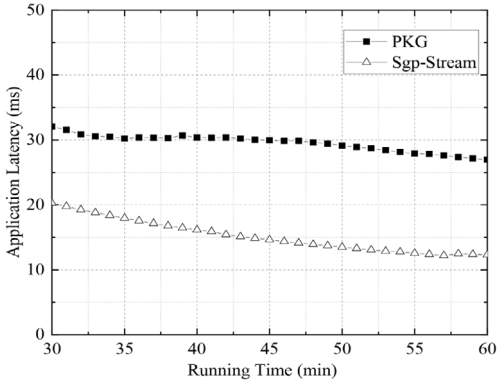


(b) PKG strategy

Fig. 12. Capacity of Bolt A under fluctuating input rates.



(a) RAS strategy



(b) PKG strategy

Fig. 13. Application latency of WordCount under the data rates of 60 000 tuples/s.

has a lower average latency than RAS. Meanwhile, we also conduct comparative experiments between PKG and Sgp-Stream, and observe that Sgp-Stream also has a lower average latency than PKG.

We then increase the input from 6000 tuple/s to the rate of 60 000 tuple/s. At this point, the parallelism is no longer suitable for the current data input rate, resulting in the average CPU utilization of the critical node exceeding the set threshold, so the parallelism adjustment mechanism is triggered. After a period of adjustment, Sgp-Stream still has lower application latency and takes less time to enter a steady state. As shown in Fig. 11, the application latency of Sgp-Stream and of RAS are 11.656 ms and 18.951 ms, respectively. As the input rate increases, the system requires more resources and Sgp-Stream can adapt to the changes of input through multi-layer coordination, therefore achieving a lower average latency. Similarly, Sgp-Stream has a lower average latency than PKG.

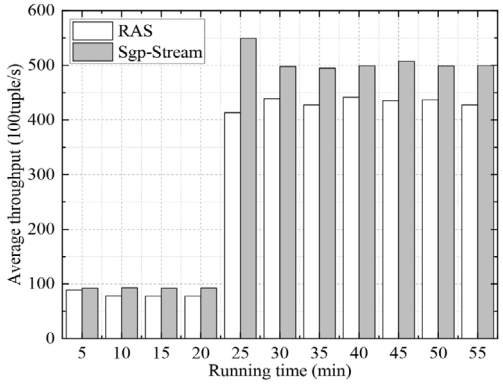
As shown in Fig. 12, when the input rate increases from 6000 tuple/s to 60 000 tuple/s at the 25th min, the capacity of Bolt A in Word-count topology changes over time. Under the rate of 6000 tuple/s, the capacities of Bolt A in Sgp-Stream and RAS are both in [0.01, 0.2]. However, when the input rate increases at the 25th min, the capacities of Bolt A in Sgp-Stream and RAS are 0.35 and 0.7 respectively. As shown in Figs. 12(a) and 12(b), under different strategies, the change trend of Bolt A's capacity is highly similar. It can be seen that with the increase of input rate, the capacity of Bolt A increases suddenly, and then gradually decreases until it stabilizes. This may be because the input rate exceeds the processing capacity of Bolt A, causing the processing delay to increase and then plateau as the processed tuples reach a bottleneck. When the processing delay exceeds a certain threshold, the system discards tuples that have been waiting for an extended period and categorizes them as failed tuples. As a result, the number of tuples processed by Bolt A decreases, resulting in a gradual decrease in its processing capacity. However, the elastic scaling module can adjust the parallelism of Bolt A in time, thus the capacity of Bolt A under Sgp-Stream is kept at [0.2, 0.4], compared to those under RAS and PKG.

Under a steady input rate of 6000 tuple/s, Sgp-Stream has a lower average application latency compared to the ResourceAware Scheduler of Storm. As shown in Fig. 13, during the period of 30 min to 60 min, the average latency of Sgp-Stream and of RAS in the stable phase are 10.352 ms and 16.140 ms, respectively, under the condition that the required capacity of the entire topology keeps stable and satisfied. The average latency of Sgp-Stream is lower than those of RAS and PKG under a high and steady rate.

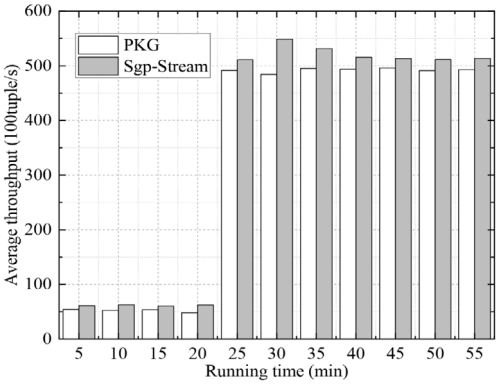
6.3. System throughput

System average throughput is one of the important indicators to measure the performance of a stream computing system. It is estimated based on the number of output tuples per second. The higher the average throughput is, the better the processing performance the system has. We use the Storm UI to retrieve the average throughput, which is measured periodically (every 60 s) at runtime. The input rate used in the experiments for application latency testing does not reach the upper limit of the system throughput. To observe the effectiveness of Sgp-Stream, we further increase the input rate to test the system throughput.

When we change the input rate from 10 000 tuples/s to 70 000 tuples/s, the system throughput also increases accordingly. As shown in Fig. 14, when the input rate has not changed (Bolt A is not a bottleneck), the average throughput of Sgp-Stream and RAS are 9289 tuples/s and 7808 tuples/s, respectively. When the input rate increases

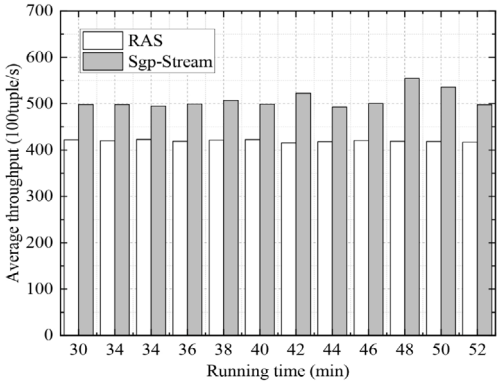


(a) RAS strategy

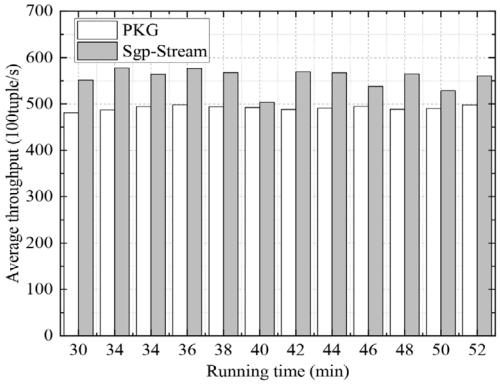


(b) PKG strategy

Fig. 14. System average throughput of WordCount under different input rates.



(a) RAS strategy



(b) PKG strategy

Fig. 15. System average throughput of WordCount under input rates of 70 000 tuples/s within [30 min, 60 min].

rapidly (Bolt A becomes a bottleneck), the average throughput of Sgp-Stream and RAS are 49 937 tuples/s and 44 135 tuples/s, respectively. This shows that under the input rate of 10 000 tuples/s, the system is underloaded and Sgp-Stream outperforms RAS in terms of throughput, but the difference is not significant. However, under the input rate of 70 000 tuples/s, the system has higher load and the average throughput of Sgp-Stream is significantly higher than those of RAS and PKG.

When the input rate is kept 70 000 tuples/s, the average throughput of the system will gradually stabilize to a reasonable level as the time elapses. As shown in Fig. 15, the average throughput of Sgp-Stream and RAS from 30 to 60 min is 49 880 tuples/s and 41 814 tuples/s, respectively. This shows that Sgp-Stream has better average system throughput than RAS strategy and PKG strategy under a steady input rate of 70 000 tuples/s.

6.4. Resource utilization rate

The average CPU utilization of compute nodes in the cluster is also used as one of the indicators to measure system performance. When the system is running stream applications, higher CPU usage in the cluster's compute nodes generally correlates with better performance indicators. The system consumes cluster resources to process data stream. The average CPU utilization measures whether the system effectively uses the cluster resources. In the following experiments, we use the "top" command to periodically measure the average CPU utilization of multiple heavily loaded compute nodes in the cluster.

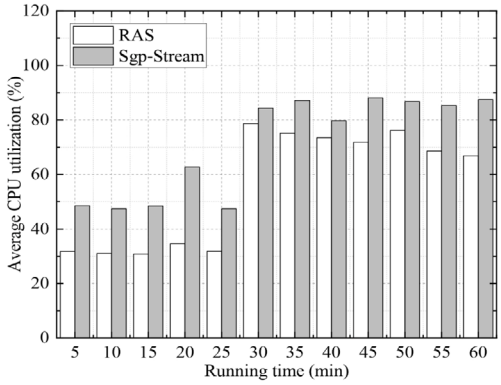
When the input rate is set to 10 000 tuples/s, as shown in Fig. 16, the average CPU utilization of Sgp-Stream is higher than those of RAS and PKG during the first 25 min. When the CPU load of compute nodes stays in [25,40], in order to better use the resources, Sgp-Stream aggregates the load from the compute nodes with low CPU utilization into a smaller number of compute nodes without overloading them. This "movement" will reduce the communication cost between nodes and improve the average CPU utilization of the cluster. However, RAS and PKG cannot dynamically adjust the number of workers and parallelism configuration used in the topology, resulting in a waste of resources and low utilization.

When the input rate is set to 70 000 tuples/s, as shown in Fig. 16, the average CPU utilization of RAS and PKG is consistently lower than Sgp-Stream from 30 min to 60 min. This shows that Sgp-Stream can efficiently utilize the resources of each compute node in the cluster, and maintain a higher average CPU utilization, which not only improves the system performance, but also greatly improves the average CPU utilization of the cluster.

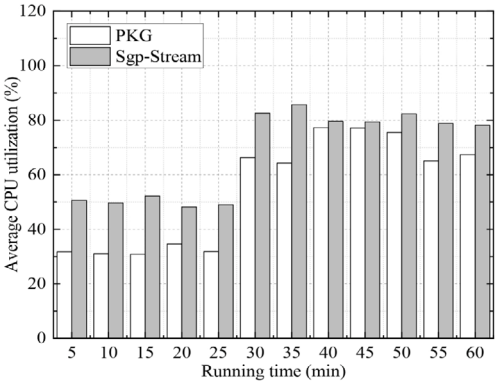
When the input rate remains 70 000 tuples/s, the average CPU utilization of the system gradually stabilizes to a certain level as the time passes by. As shown in Fig. 17, Sgp-Stream stabilizes the average CPU utilization in [75,85] between 38 min and 60 min. Compared to the ResourceAware Scheduler strategy, Sgp-Stream makes better use of resources of the cluster.

7. Conclusions and future work

We evaluate the impact of multiple factors on system performance across multiple dimensions and analyze their interactions. Our analysis suggests that better overall performance can be achieved by coordinating multiple factors at multiple levels. We further establish quantitative models for stream applications represented by directed acyclic graphs (DAG), multi-dimensional featured data stream, data center resources, and latency & throughput performance.

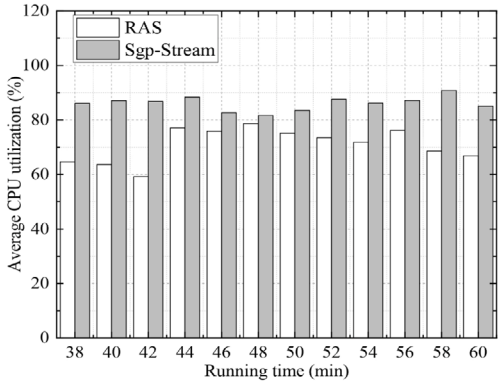


(a) RAS strategy

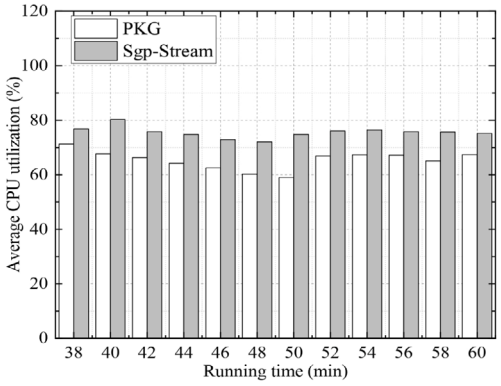


(b) PKG strategy

Fig. 16. System average CPU utilization of WordCount under different input rates.



(a) RAS strategy



(b) PKG strategy

Fig. 17. System average CPU utilization of WordCount under input rates of 70 000 tuples/s within [38 min 60 min].

Then, we propose a framework named Sgp-Stream by orchestrating scheduling, grouping and parallelism. The objective of this paper is to achieve optimal system performance by employing simple yet effective algorithm for each level of strategy — scheduling, grouping, and parallelism scaling within the Sgp-Stream framework. Each strategy aims for minimal cost while optimizing local performance, yet there is room for improving the global optimization.

We adopt strategies for runtime-aware data stream grouping based on smooth weighted polling, elastic adaptive scheduling based on Linear Deterministic Greedy (LDG) and elastic scaling based on Gradient Descent (GD) to handle fluctuating data streams from different levels. Through the coordination of different levels of improvement schemes, the system performance can continuously reach its optimal state.

Experimental results show that the performance under Sgp-Stream is significantly better than the ResourceAware Scheduler and PKG strategy, in terms of application latency, throughput and resource utilization.

In the future, we will include a prediction module to predict the change of input rate, and apply artificial intelligence algorithms such as machine learning or deep learning to further optimize system performance globally. Furthermore, we will investigate the relationship between high capacity, input rate, and system performance. The complex relationship between resource utilization and power consumption, as well as the issue of maintaining stateful operators' state consistency during load redirection, will be investigated within the Sgp-Stream framework as part of our research agenda.

CRediT authorship contribution statement

Dawei Sun: Conceptualization, Methodology, Validation, Writing – original draft, Funding acquisition. **Haiyang Chen:** Formal analysis, Methodology, Investigation, Writing, Data curation. **Shang Gao:** Formal analysis, Investigation, Writing – review & editing. **Rajkumar Buyya:** Methodology, Writing – review & editing, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The data that has been used is confidential.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grant No. 62372419; the Fundamental Research Funds for the Central Universities, China under Grant No. 265QZ2021001; Melbourne-Chindia Cloud Computing (MC3) Research Network, Australia.

References

Borkowski, M., Hochreiner, C., & Schulte, S. (2018). Moderated resource elasticity for stream processing applications. *Lecture Notes in Computer Science, 10659*, 5–16.

Cao, H., Wu, C. Q., Bao, L., Hou, A., & Shen, W. (2020). Throughput optimization for storm-based processing of stream data on clouds. *Future Generation Computer Systems, 112*, 567–579.

Cardellini, V., Nardelli, M., & Luzi, D. (2016). Elastic stateful stream processing in storm. In *2016 international conference on high performance computing & simulation* (pp. 583–590).

Chen, F., Wu, S., & Jin, H. (2018). Network-aware grouping in distributed stream processing systems. *Lecture Notes in Computer Science*, *11334*, 3–18.

Chen, H., Zhang, F., & Jin, H. (2017). Popularity-aware differentiated distributed stream processing on skewed streams. In *2017 IEEE 25th international conference on network protocols* (pp. 1–10).

Cugola, G., & Margara, A. (2012). Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys*, *44*(3), 1–62.

De Matteis, T., & Mencagli, G. (2017). Elastic scaling for distributed latency-sensitive data stream operators. In *2017 25th euromicro international conference on parallel, distributed and network-based processing* (pp. 61–68).

Dias de Assunção, M., da Silva Veith, A., & Buyya, R. (2018). Distributed data stream processing and edge computing: a survey on resource elasticity and future directions. *Journal of Network and Computer Applications*, *103*, 1–17.

Duan, W., & Zhou, L. (2020). Task scheduling optimization based on firefly algorithm in storm. In *2020 IEEE 10th international conference on electronics information and emergency communication* (pp. 150–154).

Farrokh, M., Hadian, H., Sharifi, M., & Jafari, A. (2022). SP-ant: An ant colony optimization based operator scheduler for high performance distributed stream processing on heterogeneous clusters. *Expert Systems with Applications*, *191*, Article 116322.

Flink (2024). https://flink.apache.org/.

Fu, T. Z., Ding, J., Ma, R. T., Winslett, M., Yang, Y., & Zhang, Z. (2015). DRS: Dynamic resource scheduling for real-time analytics over fast streams. In *2015 IEEE 35th international conference on distributed computing systems* (pp. 411–420).

Fu, T. Z. J., Ding, J., Ma, R. T. B., Winslett, M., Yang, Y., & Zhang, Z. (2017). DRS: Auto-scaling for real-time stream analytics. *IEEE-ACM Transactions on Networking*, *25*(6), 3338–3352.

García-Vico, Á. M., Carmona, C., González, P., & del Jesus, M. J. (2021). A cellular-based evolutionary approach for the extraction of emerging patterns in massive data streams. *Expert Systems with Applications*, *183*, Article 115419.

Govindarajan, K., Kamburugamuve, S., Wickramasinghe, P., Abeykoon, V., & Fox, G. (2017). Task scheduling in big data - review, research challenges, and prospects. In *2017 ninth international conference on advanced computing* (pp. 165–173).

Herodotou, H., Odysseos, L., Chen, Y., & Lu, J. (2022). Automatic performance tuning for distributed data stream processing systems. In *2022 IEEE 38th international conference on data engineering* (pp. 3194–3197).

Karimov, J., Rabl, T., Katsifodimos, A., Samarev, R., Heiskanen, H., & Markl, V. (2018). Benchmarking distributed stream data processing systems. In *2018 IEEE 34th international conference on data engineering* (pp. 1507–1518).

Kombi, R. K., Lumineau, N., & Lamarre, P. (2017). A preventive auto-parallelization approach for elastic stream processing. In *2017 IEEE 37th international conference on distributed computing systems* (pp. 1532–1542).

Li, C., & Zhang, J. (2017). Real-time scheduling based on optimized topology and communication traffic in distributed real-time computation platform of storm. *Journal of Network and Computer Applications*, *87*, 100–115.

Liu, X., & Buyya, R. (2020). Resource management and scheduling in distributed stream processing systems: A taxonomy, review, and future directions. *ACM Computing Surveys*, *53*(3), 1–41.

Liu, S., Weng, J., Wang, J. H., An, C., Zhou, Y., & Wang, J. (2019). An adaptive online scheme for scheduling and resource enforcement in storm. *IEEE/ACM Transactions on Networking*, *27*(4), 1373–1386.

Lombardi, F., Aniello, L., Bonomi, S., & Querzoni, L. (2018). Elastic symbiotic scaling of operators and resources in stream processing systems. *IEEE Transactions on Parallel and Distributed Systems*, *29*(3), 572–585.

Marangozova-Martin, V., De Palma, N., & El Rheddane, A. (2019). Multi-level elasticity for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, *30*(10), 2326–2337.

Muhammad, A., & Aleem, M. (2021a). A3-storm: topology-, traffic-, and resource-aware storm scheduler for heterogeneous clusters. *Journal of Supercomputing*, *77*(2), 1059–1093.

Muhammad, A., & Aleem, M. (2021b). BAN-storm: a bandwidth-aware scheduling mechanism for stream jobs. *Journal of Grid Computing*, *19*(3).

Muhammad, A., Aleem, M., & Islam, M. A. (2021). TOP-storm: A topology-based resource-aware scheduler for stream processing engine. *Cluster Computing*, *24*(1), 417–431.

Nandal, P., Bura, D., Singh, M., & Kumar, S. (2021). Analysis of different load balancing algorithms in cloud computing. *International Journal of Cloud Applications and Computing*, *11*, 100–112.

Nasir, M. A. U., Morales, G. D. F., Garcia-Soriano, D., Kourtellis, N., & Serafini, M. (2015). Partial key grouping: Load-balanced partitioning of distributed streams. *Computing Research Repository*.

Ni, X., Li, J., Yu, M., Zhou, W., & Wu, K. L. (2020). Generalizable resource allocation in stream processing via deep reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence* (pp. 857–864).

Nicoleta Tantalaki, S. S., & Roumeliotis, M. (2020). A review on big data real-time stream processing and its scheduling techniques. *International Journal of Parallel, Emergent and Distributed Systems*, *35*(5), 571–601.

Peng, B., Hosseini, M., Hong, Z., Farivar, R., & Campbell, R. (2015). R-storm: Resource-aware scheduling in storm. In *Proceedings of the 16th annual middleware conference* (pp. 149–161).

Qureshi, M. M., Chen, H., Zhang, F., & Jin, H. (2020). IPC: Resource and network cost-aware distributed stream scheduling on skewed streams. *Advanced Engineering Informatics*, *46*, Article 101165.

Ramesh, A., Rajkumar, S., & Livingston, J. L. (2021). Disaster management in smart cities using IoT and big data. *Journal of Physics: Conference Series*, *1716*(1), Article 012060.

Röger, H., & Mayer, R. (2019). A comprehensive survey on parallelization and elasticity in stream processing. *ACM Computing Surveys*, *52*(2), 1–37.

Sahni, J., & Vidyarthi, D. P. (2021). Heterogeneity-aware elastic scaling of streaming applications on cloud platforms. *Journal of Supercomputing*, *77*(9), 10512–10539.

Samza (2024). http://samza.apache.org/.

Sarathchandra, M., Karandana, C., Heenatigala, W., Dayarathna, M., & Jayasena, S. (2021). Resource aware scheduler for distributed stream processing in cloud native environments. *Concurrency Computations: Practice and Experience*, *33*(20), Article e6373.

Scolati, R., Fronza, I., El Ioini, N., Samir, A., Barzegar, H. R., & Pahl, C. (2020). A containerized edge cloud architecture for data stream processing. In *Cloud computing and services science* (pp. 150–176). Cham: Springer International Publishing.

Son, S., Im, H., & Moon, Y. S. (2021). Stochastic distributed data stream partitioning using task locality: design, implementation, and optimization. *Journal of Supercomputing*, *77*(10), 11353–11389.

Souravlas, S., Anastasiadou, S., & Katsavounis, S. (2021). More on pipelined dynamic scheduling of big data streams. *Applied Sciences*, *11*(1), 1–20.

Spark (2024). https://spark.apache.org/streaming/.

Storm (2024). http://storm.apache.org.

Sun, D., & Huang, R. (2016). A stable online scheduling strategy for real-time stream computing over fluctuating big data streams. *IEEE Access*, *4*, 8593–8607.

Sun, X., Li, B., Shi, T., Hu, Y., Yang, X., & Song, Y. (2019). Real-time processing for remote sensing satellite data based on stream computing. *Vol. 9081*, In *2019 IEEE international conference on signal, information and data processing* (pp. 1–8).

Vogel, A., Mencagli, G., Griebler, D., Danelutto, M., & Fernandes, L. G. (2021). Online and transparent self-adaptation of stream parallel patterns. In *Computing*: *vol. 105*, (pp. 1039–1057).

Wang, L., Fu, T. Z., Ma, R. T., Winslett, M., & Zhang, Z. (2019). Elasticutor: Rapid elasticity for realtime stateful stream processing. In *Proceedings. ACM-SIGMOD international conference on management of data* (pp. 573–588). abs/1711.01046.

Xie, C., Qian, L., DIng, L., & Yang, F. (2017). Adaptive topology decomposition for storm. In *2017 international conference on electrical engineering and informatics* (pp. 269–273).

Yudong, L., Yuqing, Z., & Zhangbin, Z. (2020). Service availability guarantee with adaptive automatic flow control. In *2020 IEEE world congress on services* (pp. 101–105).

Zeng, X., & Zhang, S. (2023). Parallelizing stream compression for IoT applications on asymmetric multicores. In *2023 IEEE 39th international conference on data engineering* (pp. 950–964).

Zhang, S., He, J., Zhou, A. C., & He, B. (2019). Briskstream: Scaling data stream processing on shared-memory multicore architectures. In *Proceedings of the 2019 international conference on management of data* (pp. 705–722).

Zhang, F., Yang, L., Zhang, S., He, B., Lu, W., & Du, X. (2020). Finestream: fine-grained window-based stream processing on CPU-gpu integrated architectures. In *USENIX ATC'20*, *Proceedings of the 2020 USeNIX conference on usenix annual technical conference* (pp. 633–647). USA: USENIX Association.

Zhou, Y., Liu, Y., Zhang, C., Peng, X., & Oin, X. (2020). TOSS: A topology-based scheduler for storm c1usters. In *2020 IEEE international parallel and distributed processing symposium workshops* (pp. 587–596).

Zhou, S., Zhang, F., Chen, H., Jin, H., & Zhou, B. B. (2019). FastJoin: A skewness-aware distributed stream join system. In *2019 IEEE international parallel and distributed processing symposium* (pp. 1042–1052).